NASA MARSHALL SPACE FLIGHT CENTER

Simulink Code Generation

Tutorial for generating C code from Simulink Models using Simulink Coder

José Carlos Molina Fraticelli

4/25/2012

Contents

1. Introduction

This document explains all the necessary steps in order to generate optimized C code from Simulink (Ref. [2]) Models. This document also covers some general information on good programming practices, selection of variable types, how to organize models and subsystems, and finally how to test the generated C code and compare it with data from MATLAB (Ref. [1]).

(NOTE: This guide was prepared using MATLAB/Simulink Ver. R2011b (Ref. [1]).)

2. Good Programming Practices in Simulink

Below are some guidelines to follow when creating Simulink (Ref. [2]) Models:

- Create bus objects for subsystems with many outputs and inputs. It makes organizing a model simpler by requiring less cabling and also makes management of the data contained in the bus easier when debugging as errors usually show up in a single location.
- Avoid using Mux blocks to create Bus signals. Always use a Bus Creator blocks.
- Avoid Crossing Cables when possible: There are a variety of ways to avoid crossing cables which can make debugging Simulink (Ref. [2]) models complicated. Some of these include: Organizing Subsystem Inputs to match cabling going into the subsystem, Goto and From blocks, Bus Creator blocks, Bus Selector blocks, among others.
- Determine variable types ahead of time. Simulink (Ref. [2]) defaults all data types to 64-bit doubles by default. This can sometimes be a problem for code generation as not all targets can support these types of variables and it can be hard to change these once a model is created. This will save a lot of time trying to find problematic typecasts.
- Use MATLAB (Ref. [1]) structures when working with Bus Signals.
- Determine whether a model is a sub-function or a subsystem.
- Avoid Algebraic Loops. They are problematic for code generation as stated by Mathworks. If the algebraic loop cannot be avoided use a Unit Delay Block to break up the loop. (NOTE: Verify that adding this block has not changed the output of the model.)
- Avoid using different sampling times between models. This is not supported by code generation.
- Avoid using variable-step solvers. The only valid solver for code generation is the Fixed Step Solver with a discrete time step.
- Build models from the ground up with code generation in mind.
- Test models as a standalone piece both in simulation and in the generated code and ensure it is working before adding additional models.
- Avoid using equal names for buses, data, constants and models. This can cause serious problems with code generation. Use a naming standard to clearly define what every object is or isn't.
- Rename models such as "Foo One Model"; to Foo_One_Model. This will help avoid problems with code generation.

- If a model name is too long try shortening it. Model names should not exceed 20 characters.
- Align models and objects that are part of the model using the alignment tools available in Simulink (Ref. [2]) these can be accessed by selecting two or more model objects and selecting align blocks. This can help clean up models immensely
- Make sure the warning indicator on MATLAB (Ref. [1]) function blocks is green. Try to fix all warnings pointed to by this indicator. Also always add the %#codegen pragma directive below the function name declaration and before any code.
- Don’t use Goto and From blocks between different models. Use a signal line instead.
- Avoid the use of extremely memory intensive blocks such as ‘Fuzzy Logic controllers’ if generating code for embedded platforms such as microcontrollers; as the generated code will not fit in the stack due to the high amount of floating point variables required.
- Avoid excessive unnecessary variables. Memory is at a premium in embedded hardware as opposed to a desktop computer.
- Use consistent signal names. This can help ease debugging immensely.
- Align all inputs to the left of the model and all outputs to the right of the model. This can help find problematic inputs/outputs faster.

3. Choosing Variable Data Types

In order to determine the correct data type for a variable one must determine what that variable will be used for. Simulink (Ref. [2]), by default, will generate all variables as 64-bit doubles. This can cause problems for embedded platforms with a low memory footprint such as microcontrollers which is where generated code is usually targeted to. Below are some tips in order to determine which variable type is viable for each action:

- boolean: If the variable is to be used as a ON/OFF switch then this variable type should be used
- int8: If the variable does not exceed 8 bits and is a character then this variable type should be used
- int16: If the variable does not exceed 16 bits and is an integer then this variable data type should be used
- int32: If the variable does not exceed 32 bits and is an integer then this variable data type should be used
- single: If the variable is a single precision floating point number then this variable data type should be used
- double: If the variable is a double precision floating point number then this variable data type should be used. (NOTE: This is the default variable type for Simulink (Ref. [2]))
- If the variables to be defined contain many elements then a structure must be created to house these elements. A Bus must be created from said structure and then used as the data type of the variable.

4. Hardware Implementation data type information

Below is a table listing device specific information for Hardware Implementation of the Simulation and Generated code. This information is set in the Hardware Implementation pane located in the Simulation Configuration Parameters:

Device vendor / Device type	Number of bits					Largest atomic size		Byte ordering	Rounds to	Shift right
	char	short	int	long	native	int	float			
ARM Compatible										
ARM 7/8/9/10	8	16	32	32	32	Long	Float	x	x	x
Microchip										
dsPIC	8	16	16	32	16	x	x	Little Endian	Zero	Set
Texas Instruments										
MSP430	8	16	16	32	16	x	x	Little Endian	Zero	Set

Table 1: Hardware Implementation Partial
(NOTE: For a complete Hardware Implementation table refer to Appendix A.)

5. Naming Standards

To remain consistent on which object is what, and also to make debugging of problems easier, it is important to establish a naming standard. This naming standard must be applied to models, constants, variables and buses. An example of this standard is listed below and can be used as is or with some modifications if required:

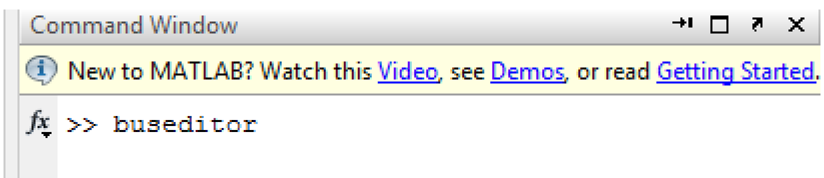
- If the object in question is a model then the name of that model should be in the form of: “Name_Model”
- If the object in question is a constant then the name of that constant should be in the form of: “Name_Constant”
- If the object in question is a bus then the name of that bus should be in the form of: “Name_Bus”

- If the object in question is a signal then the name of that signal should be in the form of: “Name_Signal”
- If the object in question is a function then the name of that signal should be in the form of: “Name_Func”
- If the object in question is a subsystem input then the name of that input should be in the form of: "Name_Input"
- If the object in question is a subsystem input then the name of that input should be in the form of: "Name_Output"

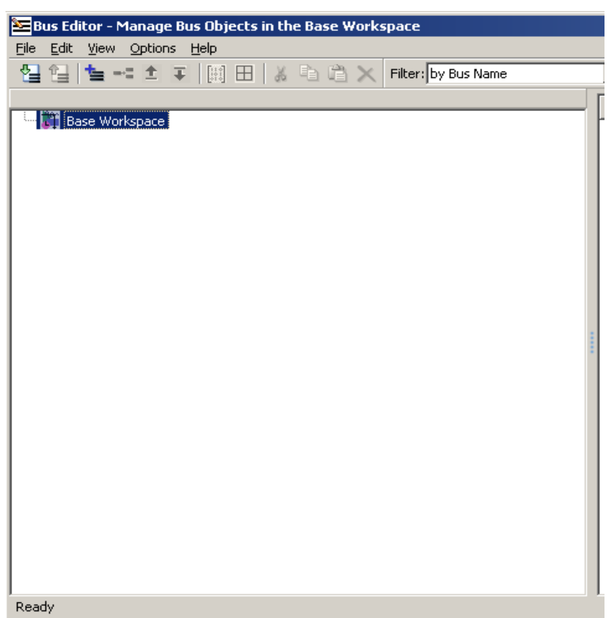
6. Creation of practice model for code generation

In this section a new practice model will be created in order to demonstrate the necessary steps to do successful code generation from any Simulink (Ref. [2]) model.

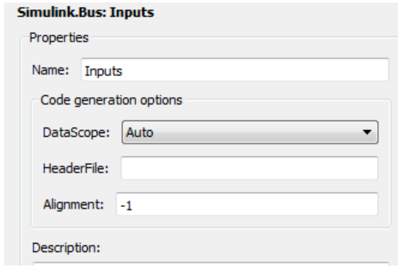
Open up MATLAB (Ref. [1]) and type the ‘buseditor’ command on the command window in order to access the Bus Editor.



Bus Editor GUI:

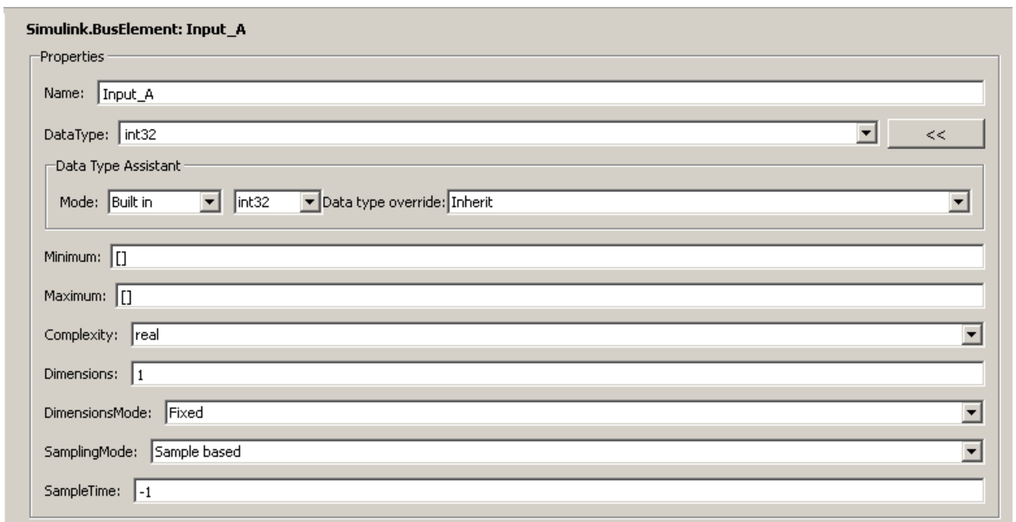


Once here click on File > Add Bus, (or alternatively Ctrl+B), while having the Base Workspace selected. A new Bus Object will be created in the MATLAB (Ref. [1]) workspace. Rename this Bus Object to Inputs by modifying the Name Field in the Bus Editor:

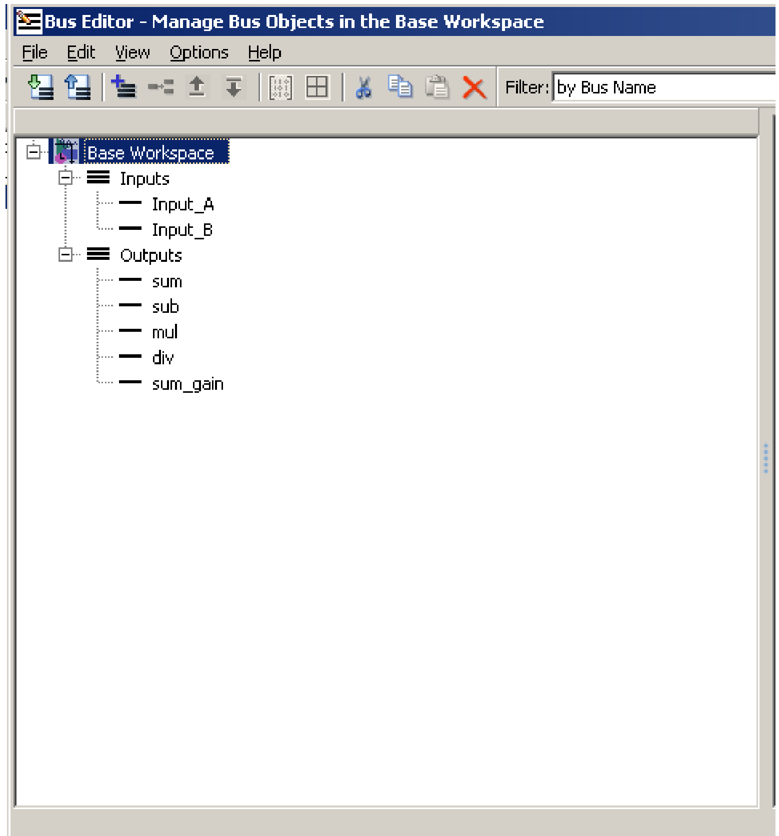


Click Apply at the bottom of this window.

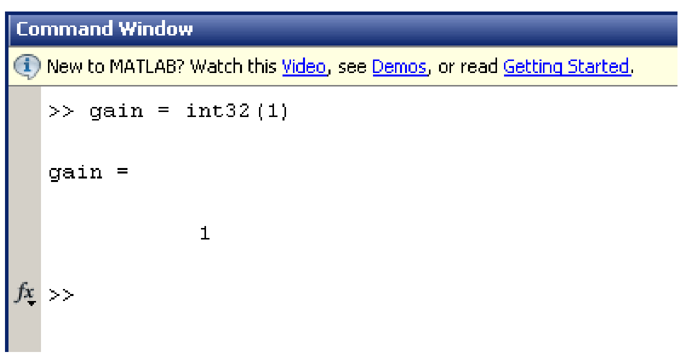
Now add two new elements to this Bus by going to File>Add/Insert Element (or alternatively Ctrl+E) while having this Bus Object selected. These elements will be called Input_A and Input_B respectively and will be of type int32:



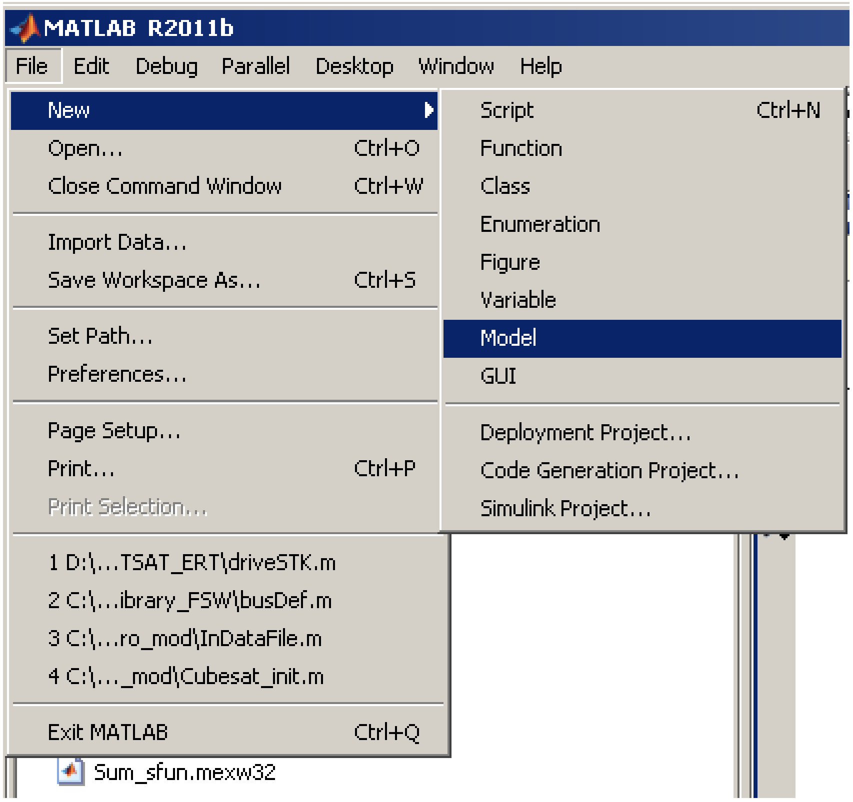
Now repeat the previous process for the Outputs Bus but this time add five elements sum of type int32, sub of type int32, mul of type int32, div of type double, and sum_gain of type int32. Once finished the Bus Editor Window should contain the following objects:



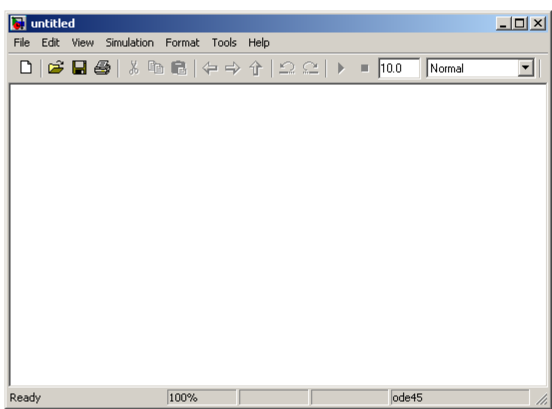
Go back to the workspace and write the following command in the command window: ‘gain = int32(1)’. You should see the following now:



Now that the workspace contains all necessary bus objects and variables with their defined data types proceed to create a new model for Simulink (Ref. [2]) by going to File>New>Model at the top toolbar:



Now this new model should be open:

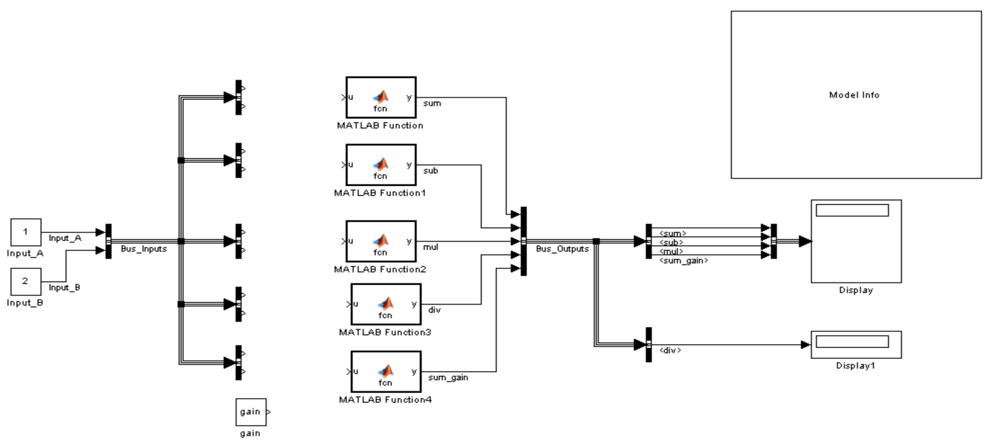


Bring up the Simulink (Ref. [2]) Library Browser and add the following blocks to the model:

- 3 Bus Creator Blocks

- 3 Constant Blocks
- 7 Bus Selector Blocks
- 2 Display Blocks
- 5 MATLAB (Ref. [1]) Function Blocks
- 1 Model Info Block

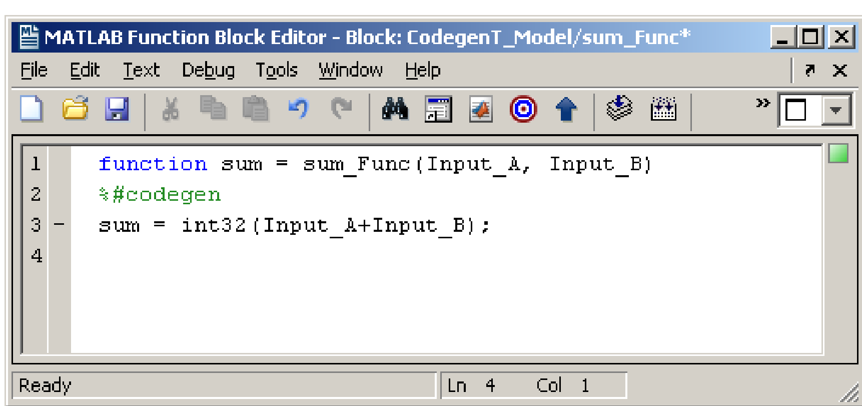
Once these are added to the model organize them in the following manner:



Make sure that the Output data types of the Input_A, Input_B and gain constants are of type int32. (NOTE: Set the gain constant value to ‘gain’)

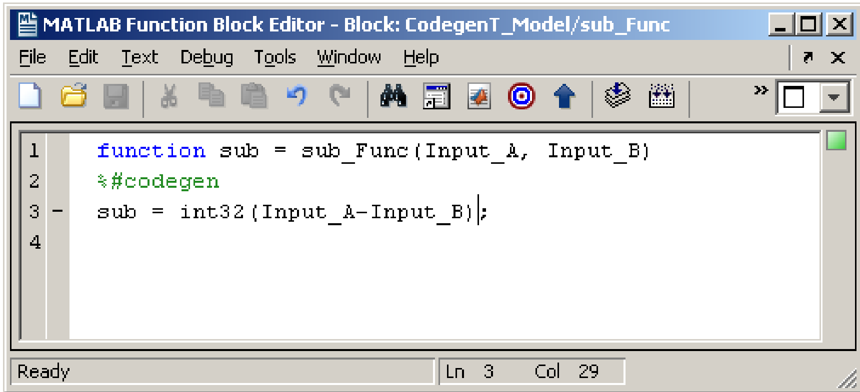
Open the first of the MATLAB (Ref. [1]) Function blocks and write the following code inside:

sum_func:

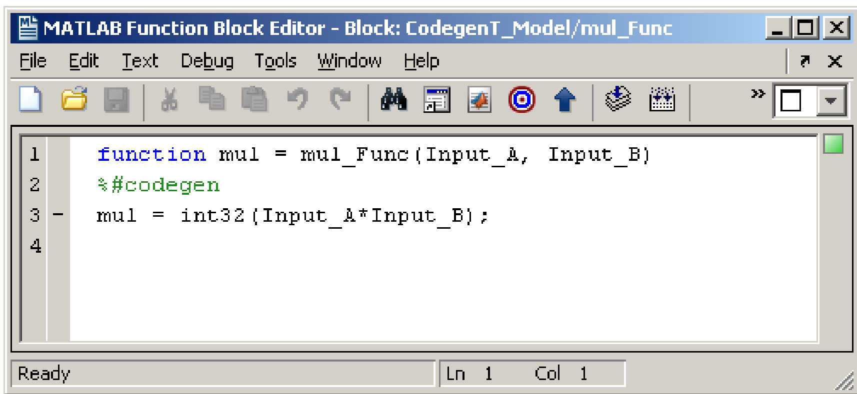


For the next four blocks write the following code in order from top to bottom:

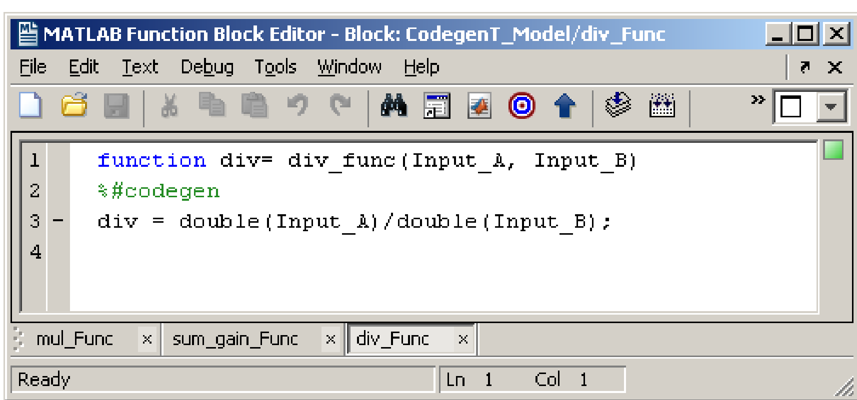
sub_func:



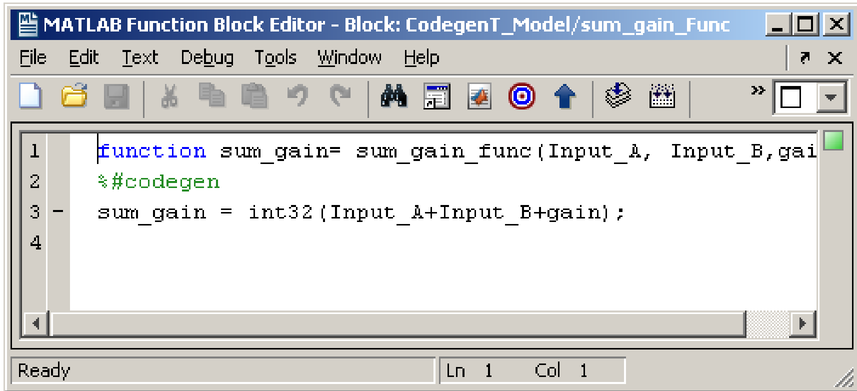
mul_func:



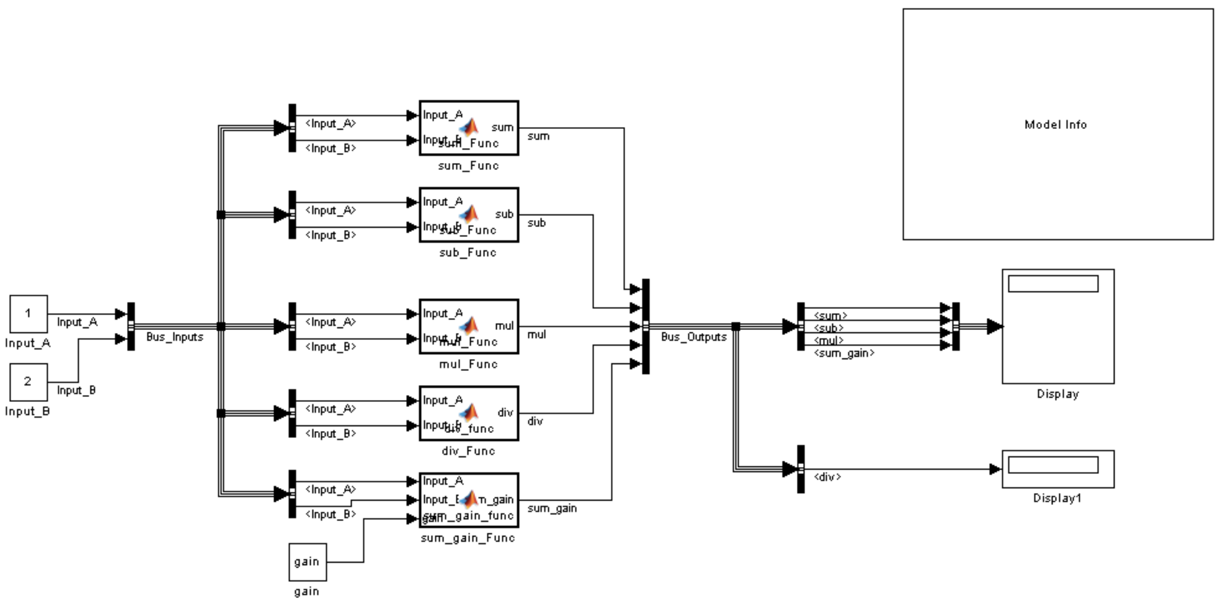
div_func:



sum_gain_func:

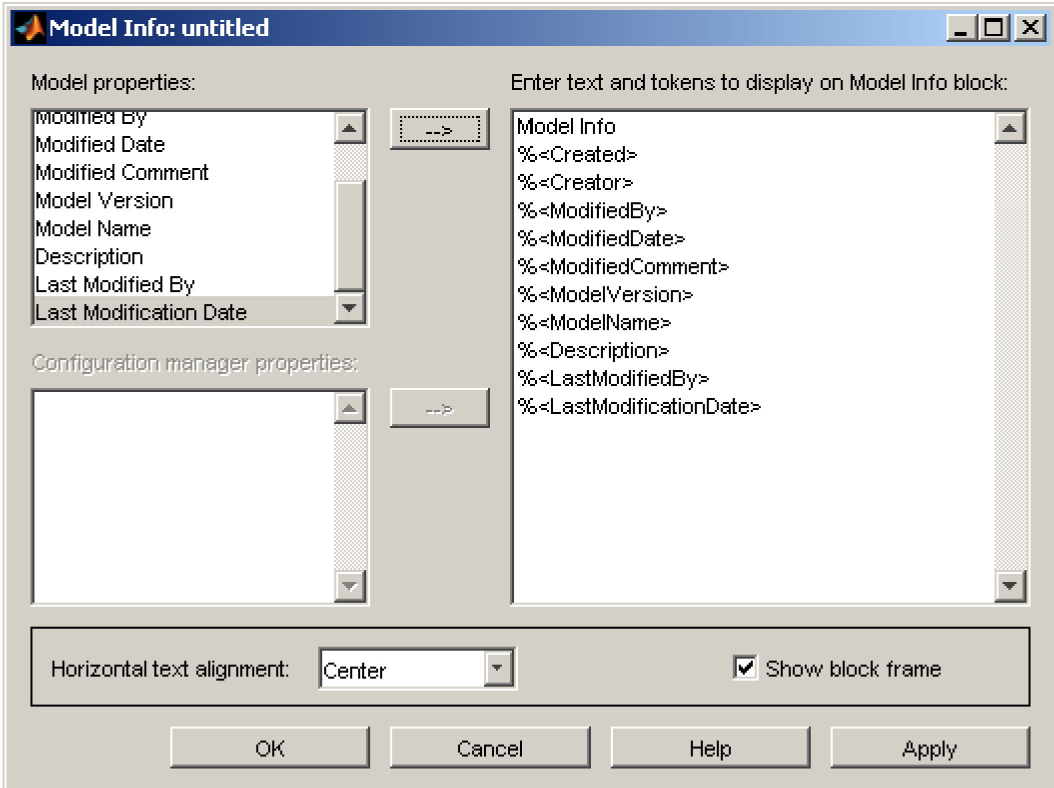


Connect the model as shown below:

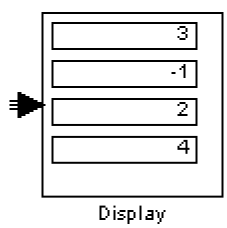


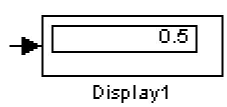
(NOTE: Make sure the First Bus Creator on the left has a data type of ‘Bus: Inputs’ and that the following Bus Creator to the right has a datatype of ‘Bus: Outputs’ as can be seen on the above. Also be sure to label all signal cables with their corresponding names)

Configure the Model Info Block with the following tokens then click Apply and OK:

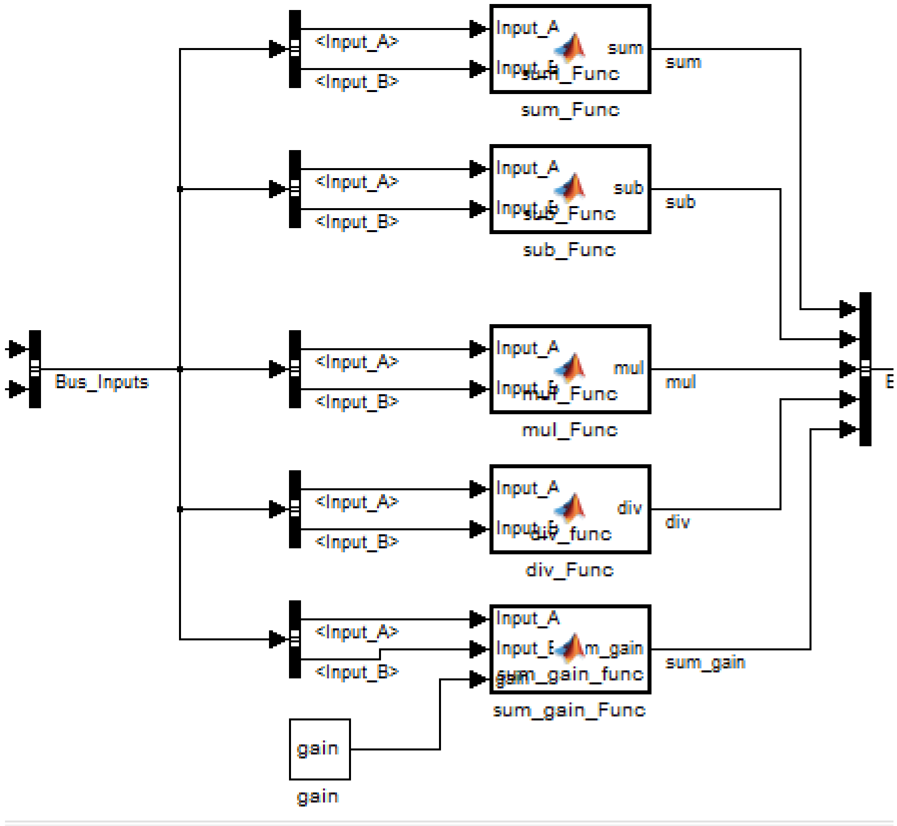


Run the model and look at the output seen on the display block; if the Input Constant values of Input_A and Input_B are set to 1 and 2 respectively then the output of the simulation should be $um = 3,\ sub = -1,\ mul = 2,\ sum_{gain} = 4\ and\ div = 0.5$:

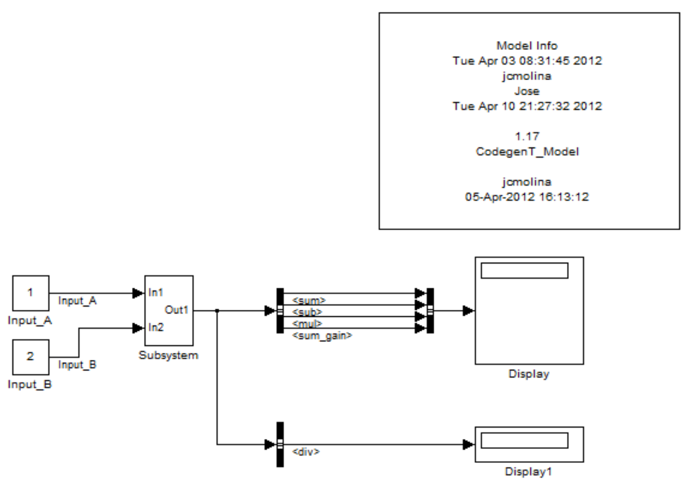




Select this section of the model and right click it. Afterwards select Create Subsystem from the popup menu:

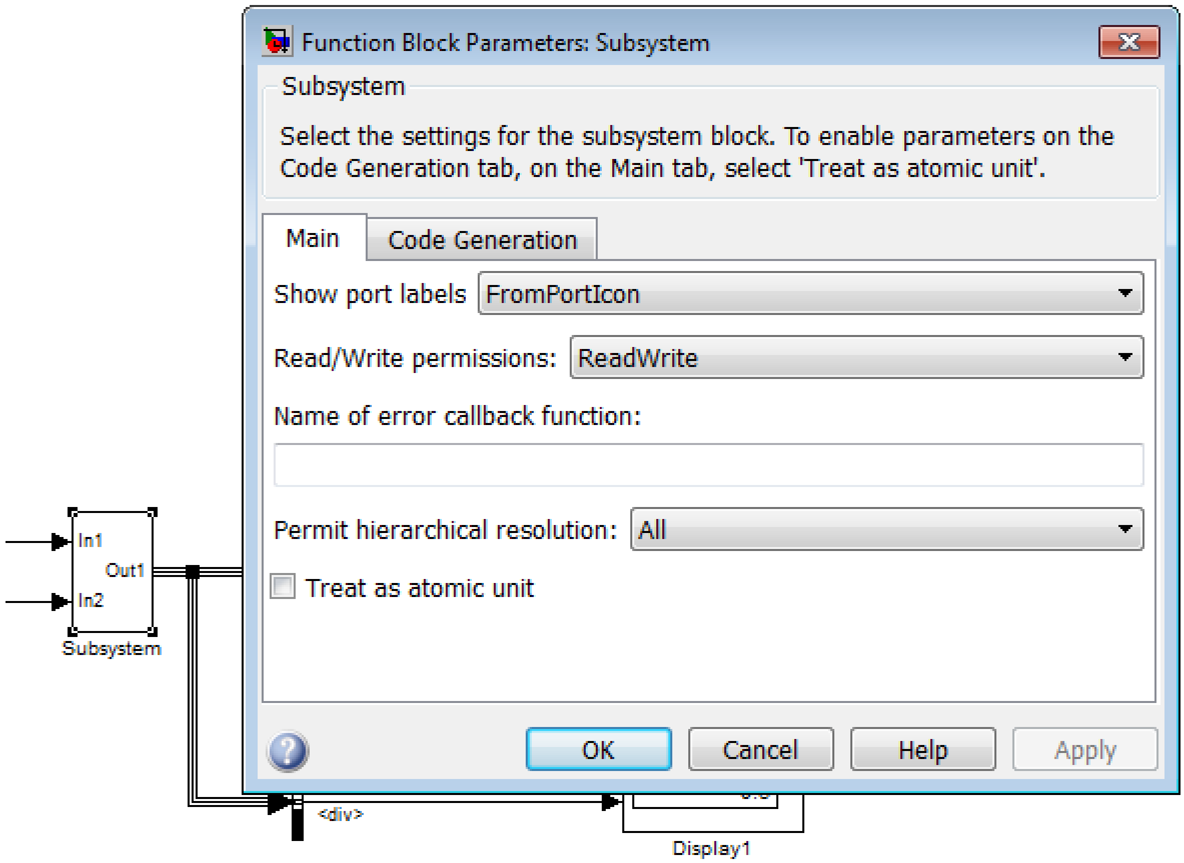


The model should now look as follows. Save this model as CodegenT_Model:

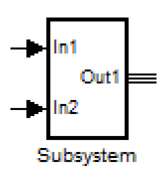


7. Preparing practice model for code generation

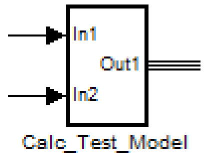
Now that the CodegenT_Model is finished and running it is time to prepare it for code generation. To start right click on the Subsystem Model and select Subsystem Parameters:



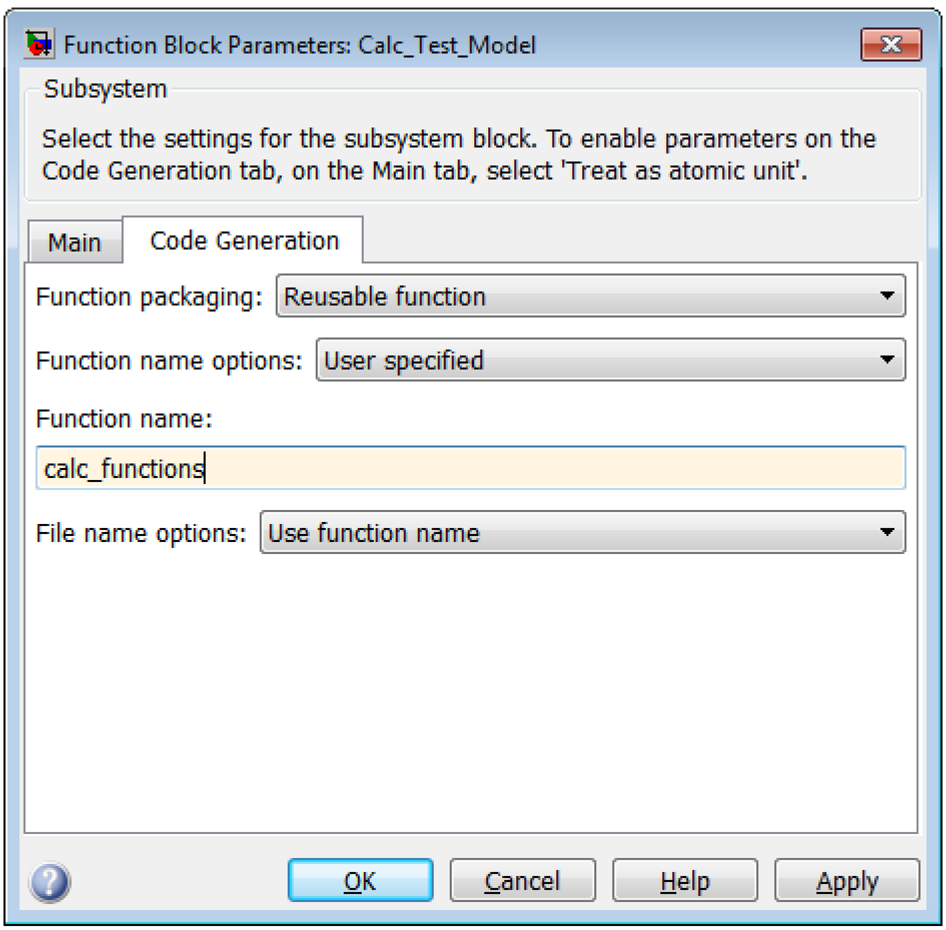
Check the box that is labeled "Treat as atomic unit"; click Apply then OK. This will change the borders of the Subsystem to a bold line indicating that the system is now atomic and will execute as a single unit inside the simulation. Systems must be atomic in order to be generated as re-usable C functions using Real-Time Workshop. The Subsystem model should now look as follows (NOTE: Changing a Subsystem to an atomic representation can change simulation output; always verify that the simulation is still working before generating code from it.):



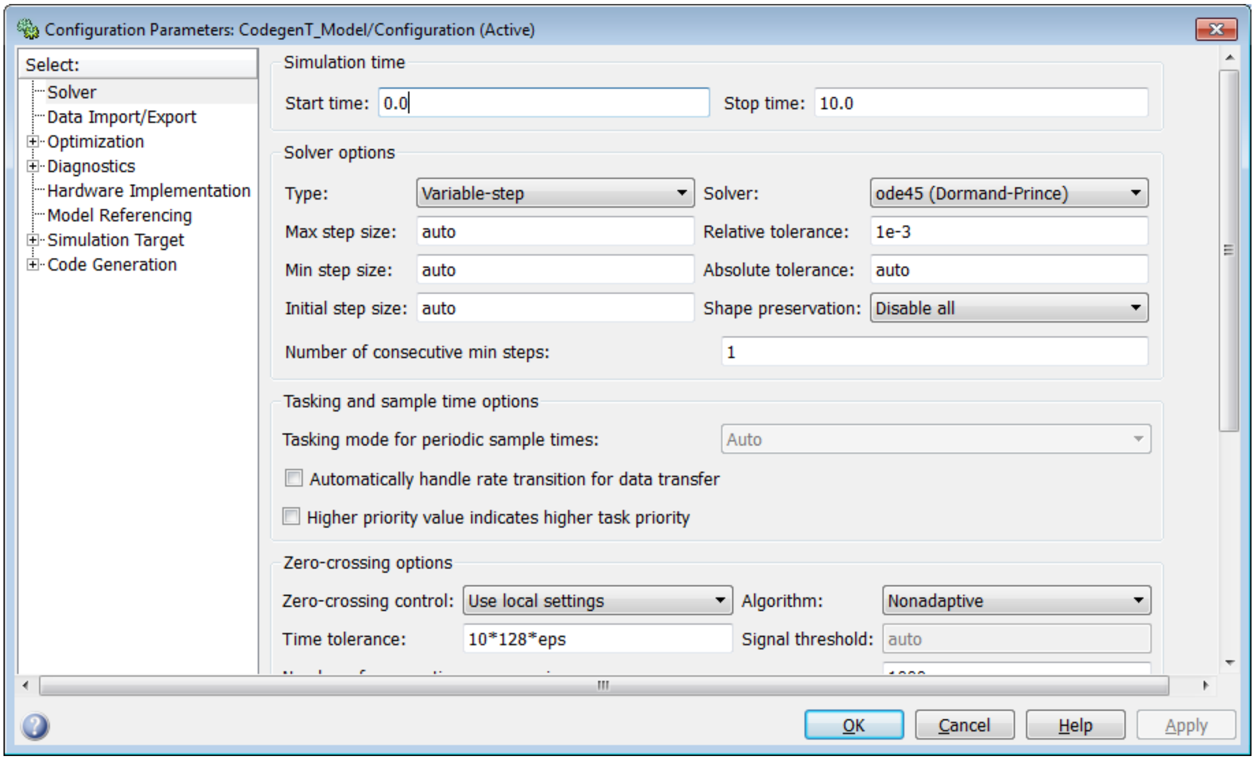
Rename the Subsystem model to Calc_Test_Model:



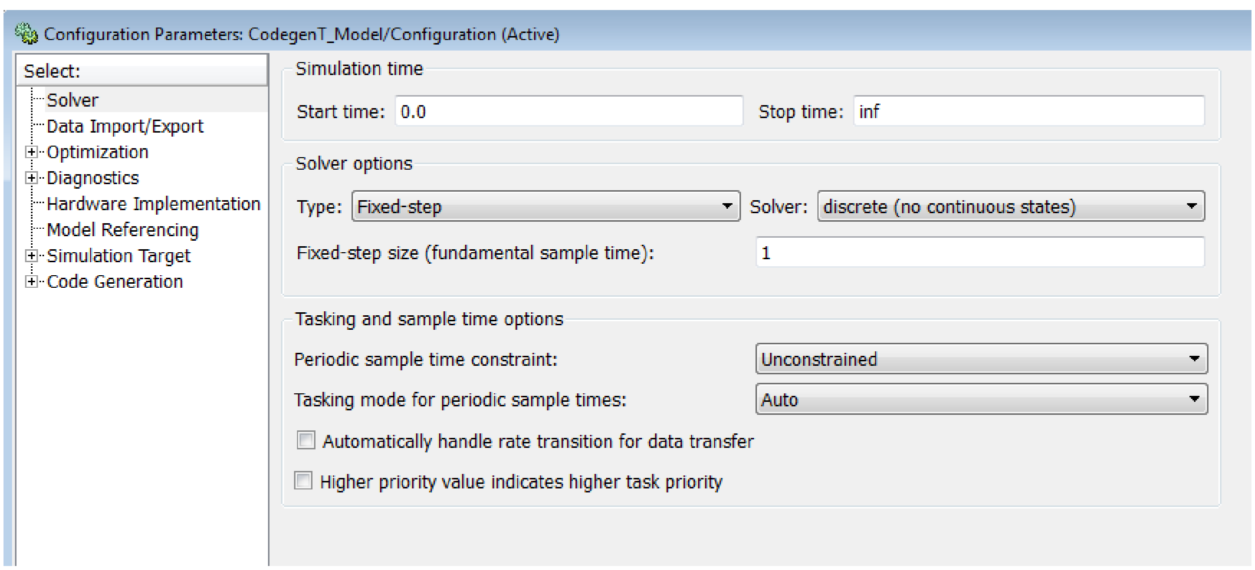
Once again right click on the model and open the Subsystem Parameters, go to the Code Generation Tab which is enabled when the Subsystem is declared as atomic and choose the "Reusable Function" option on the "Function packaging" dropdown list. Afterwards two more dropdown lists will appear "Function name options" and "File name options"; For the first dropdown list choose "User Specified" and write the following name in the provided text space "calc_functions". Afterwards choose the "Use function name" option for the "File name options" dropdown list. Finally click Apply then OK to close the subsystem parameters:



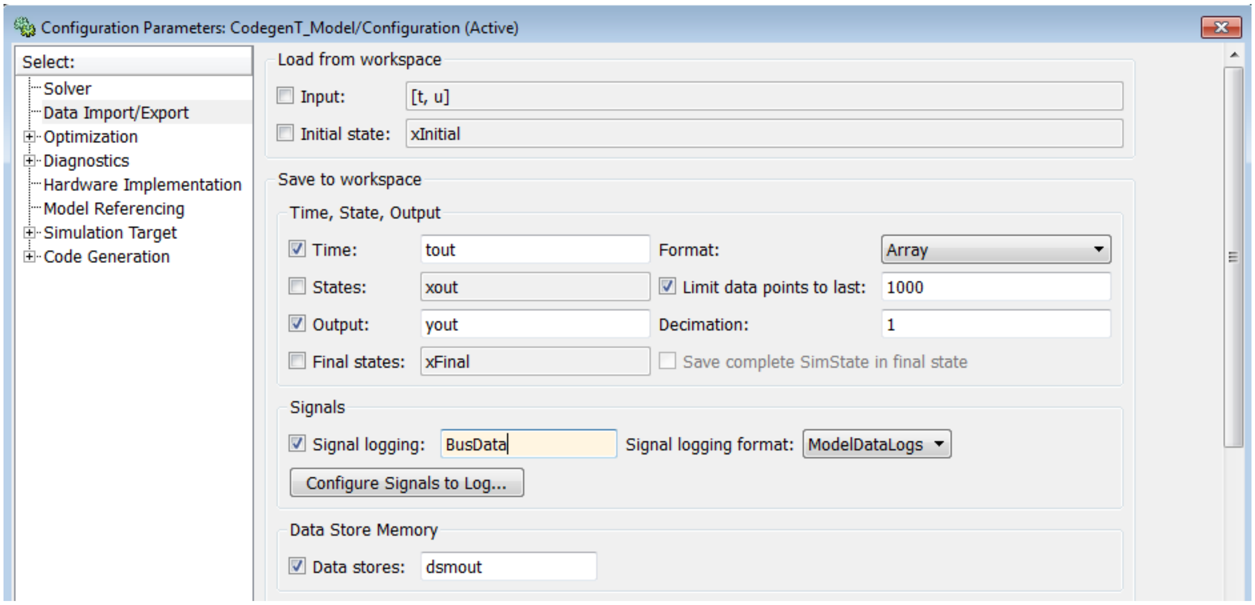
Next go to the top toolbar in Simulink (Ref. [2]) and click on Simulation > Configuration Parameters. The following window will open:



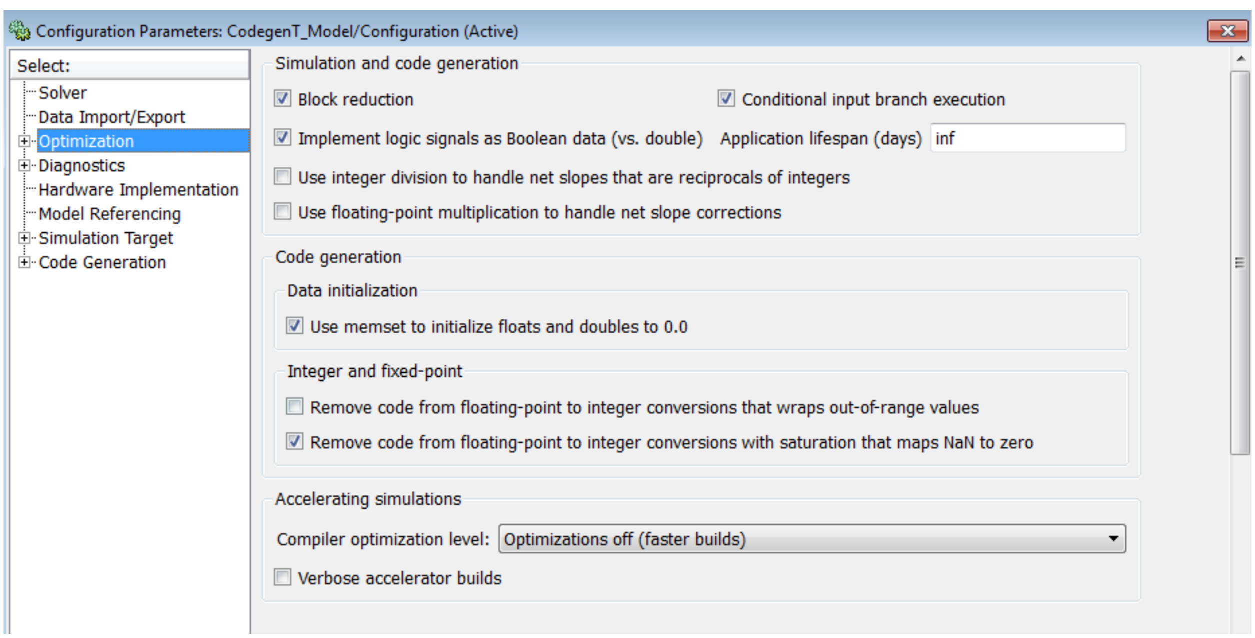
These are all the available properties for the simulation. For code generation only some of these properties need to be modified. To start off set the Solver pane options as shown below:



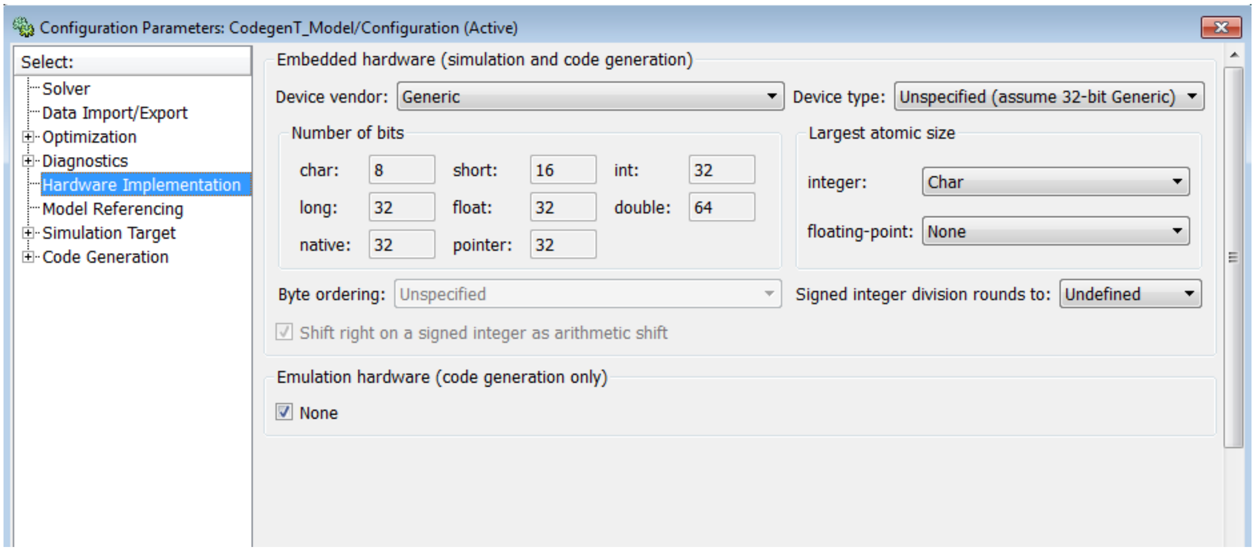
Next in Data Import/Export pane change the signal logging name to BusData (NOTE: Signal logging can be used for debugging and generating files from simulation output data):



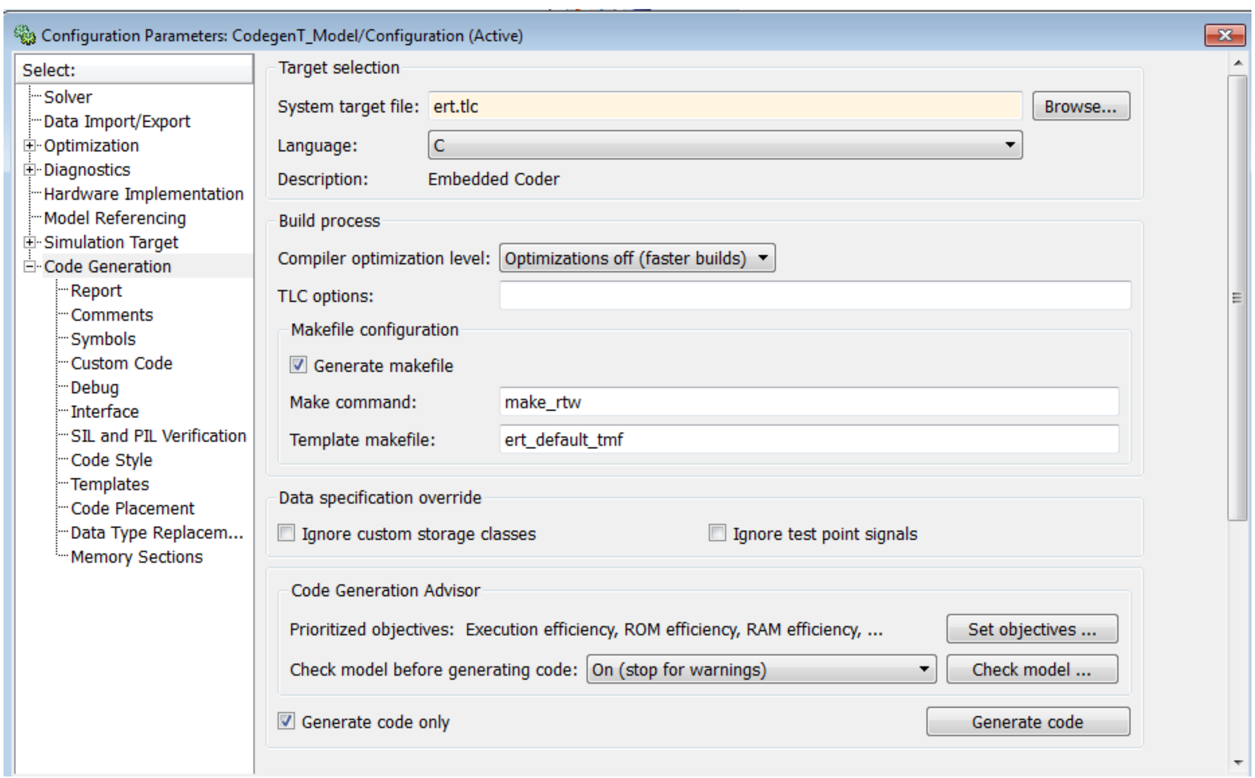
For the Optimization pane change the Application lifespan setting to inf:



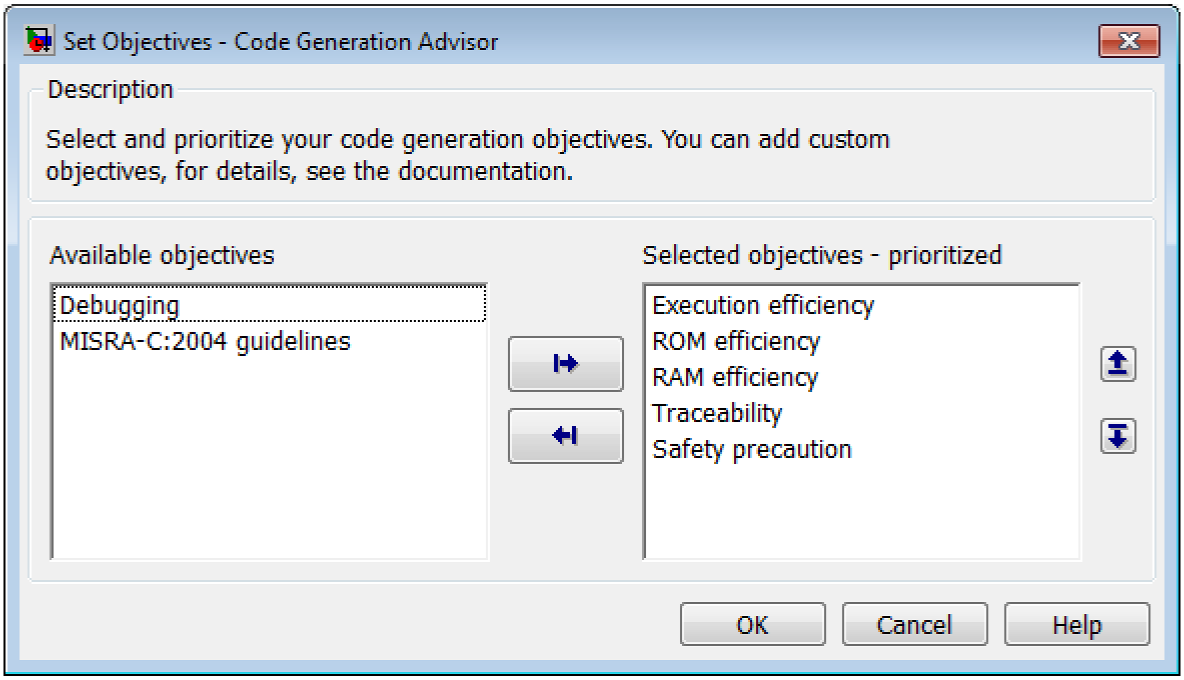
Jump over to the Hardware Implementation pane and set the settings of this pane as outlined in Table 1 based on the Hardware for which the code will be generated (NOTE: For this example Unspecified (32-bit generic) will be used):



Go to the Code Generation Pane and select the following options:

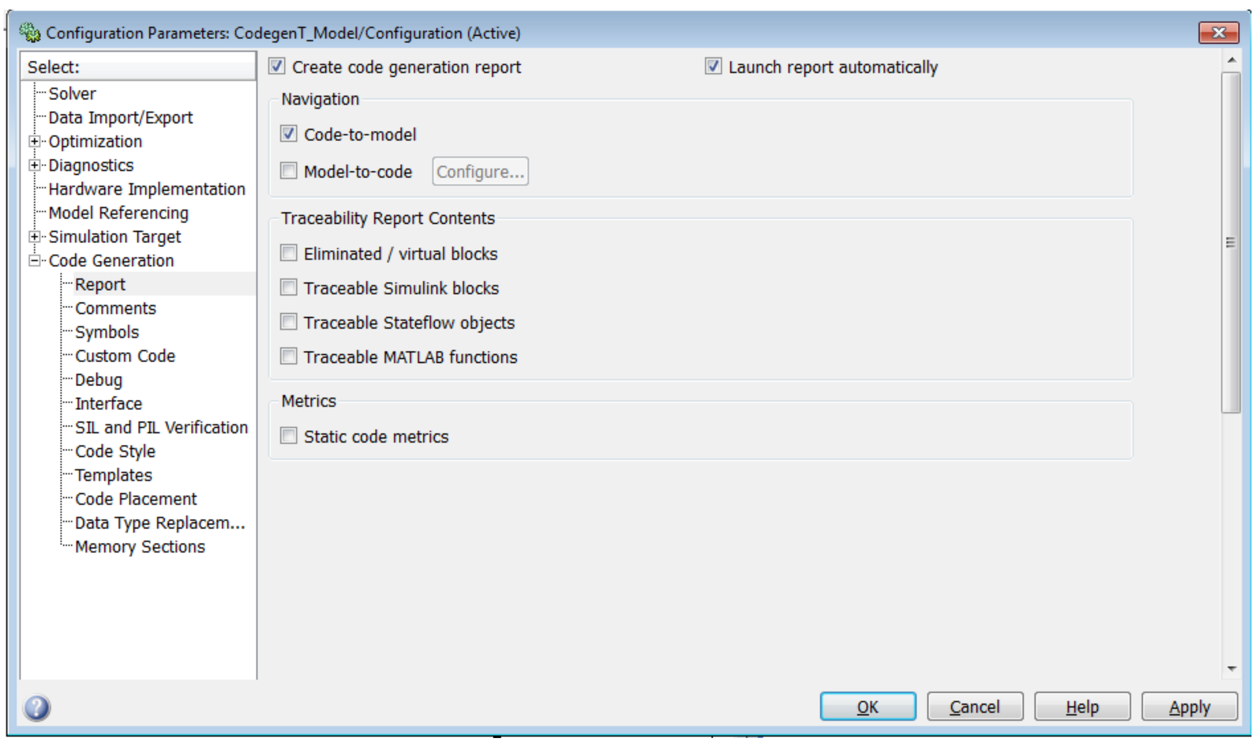


Set Objectives Pop-Up to the desired objectives:

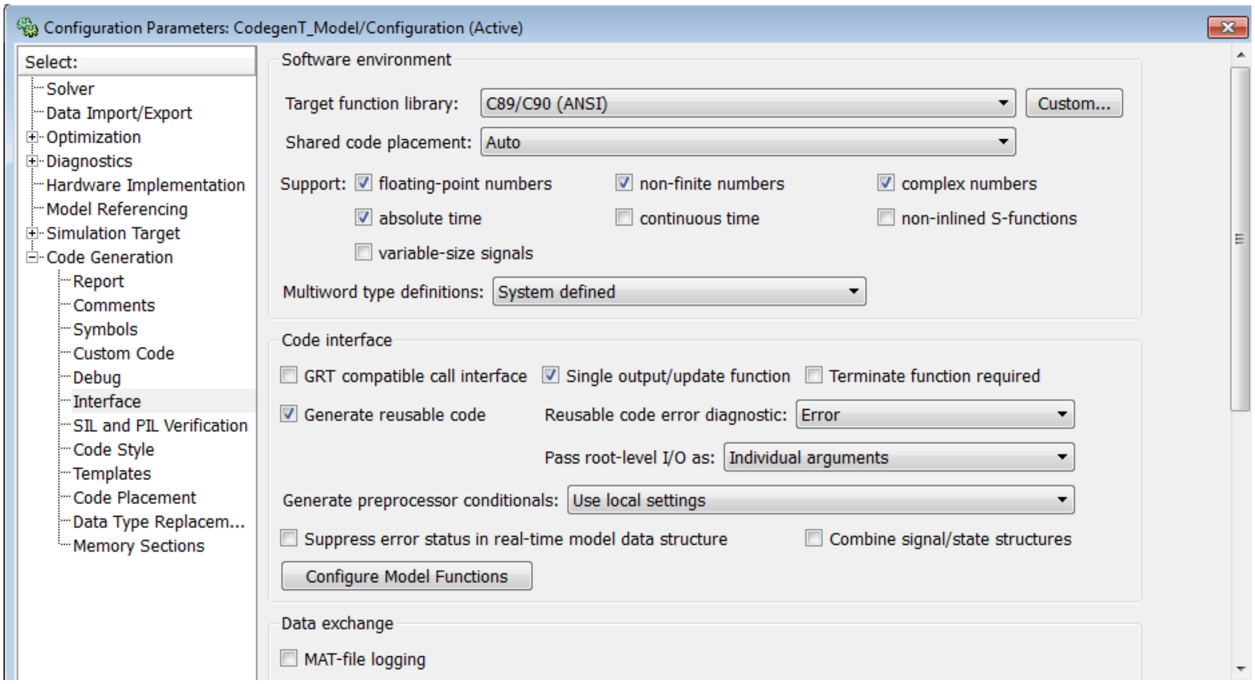


Make sure to select the correct target file based on the desired target. For this example ert.tlc Embedded Coder will be used.

Select the Report Pane and set the following options:

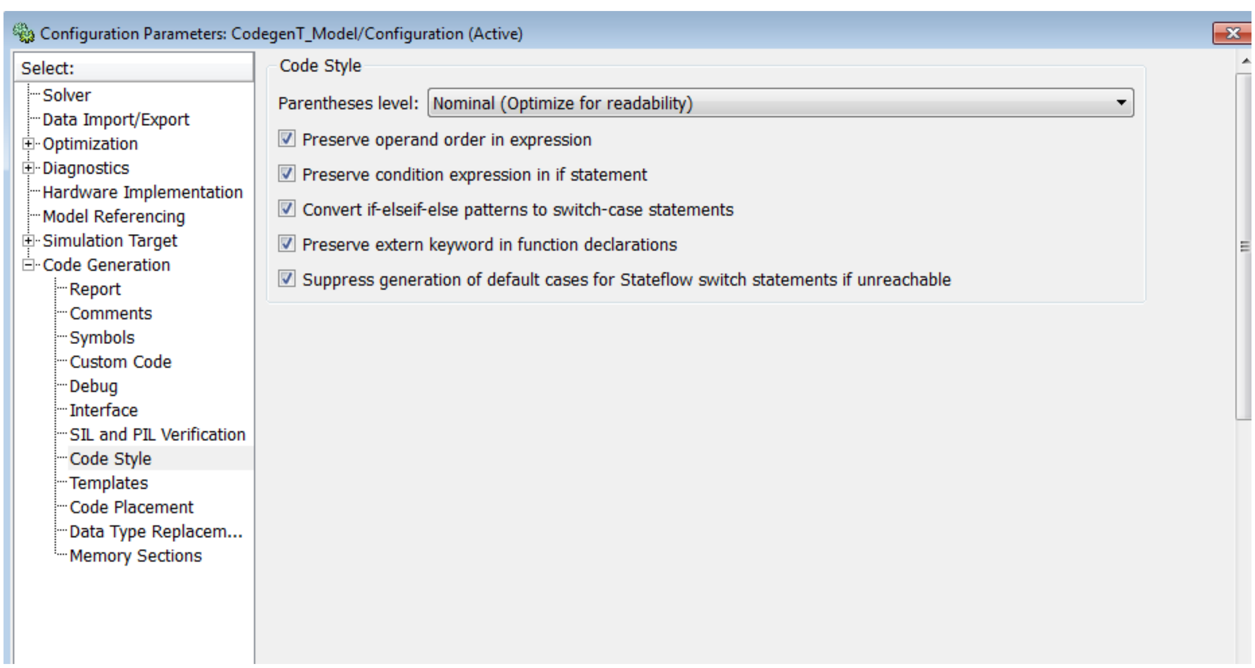


Go to the Interface pane and select the following options:

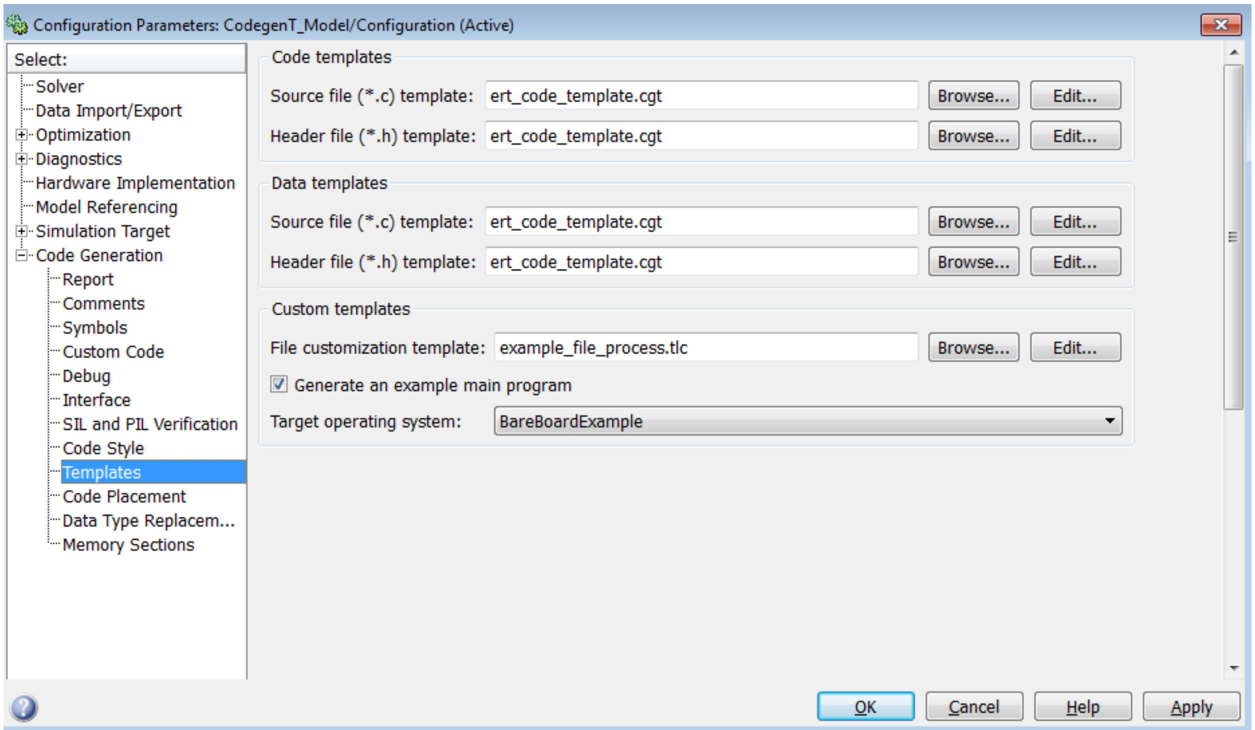


Make sure the "Generate reusable code" checkbox is checked in this pane.

Go to the Code Style pane and check all checkboxes. This is done to set the coding style as humanly readable as possible for code generation:

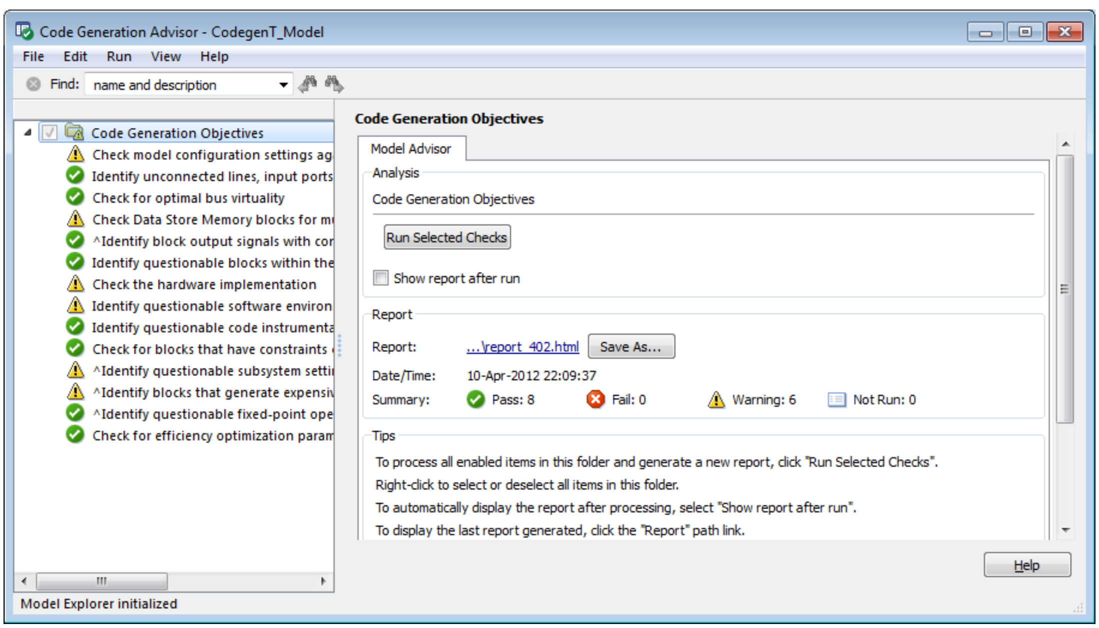


Next go to the Templates pane and set the following options:

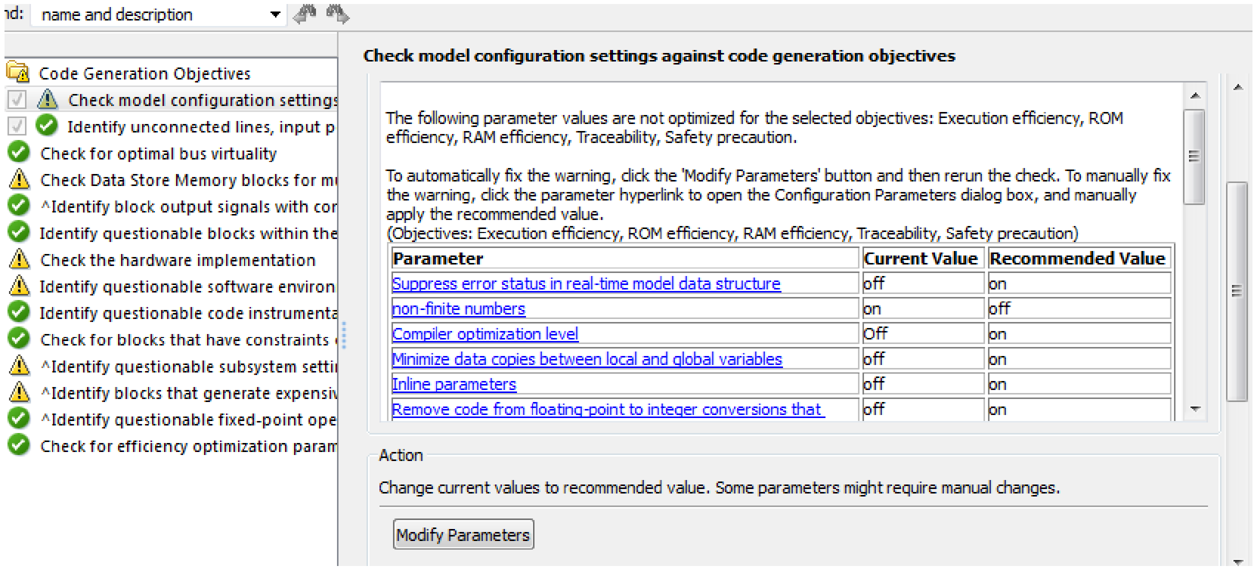


In this pane source code generation templates may be selected. This requires advanced knowledge of tlc language. Information on tlc language and files can be found in the MATLAB (Ref. [1]) help window.

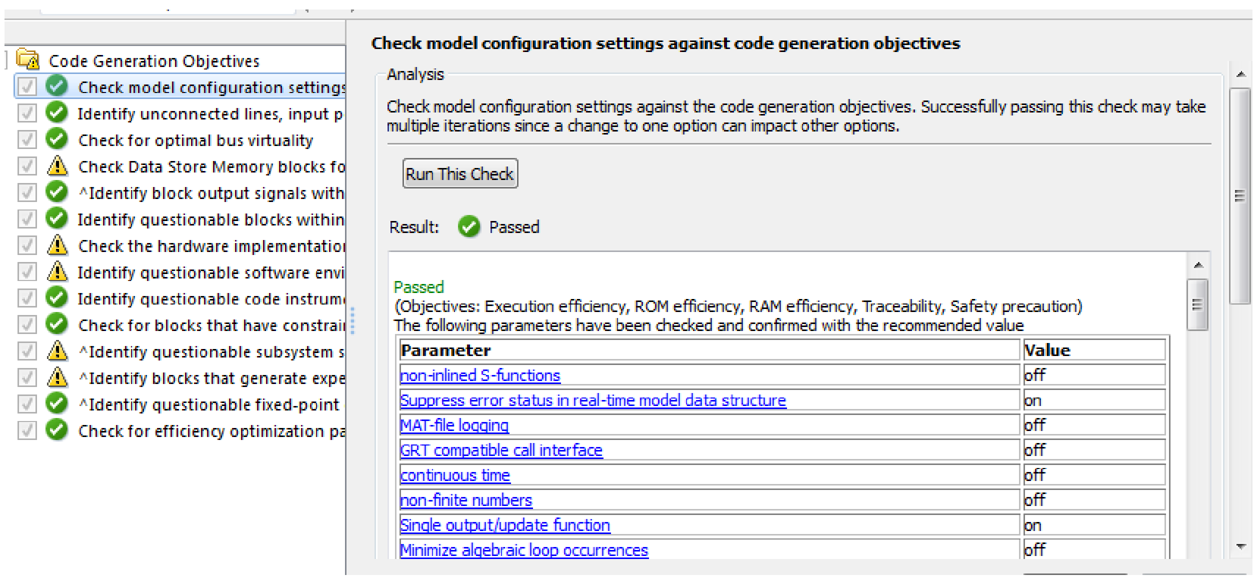
Once again return to the Code Generation pane; click Apply then click the generate code button. A model check procedure will now start to verify that all settings are correct for code generation:



Select the first warning symbol on the top of the Code Generation Objectives list and then click on Modify Parameters:

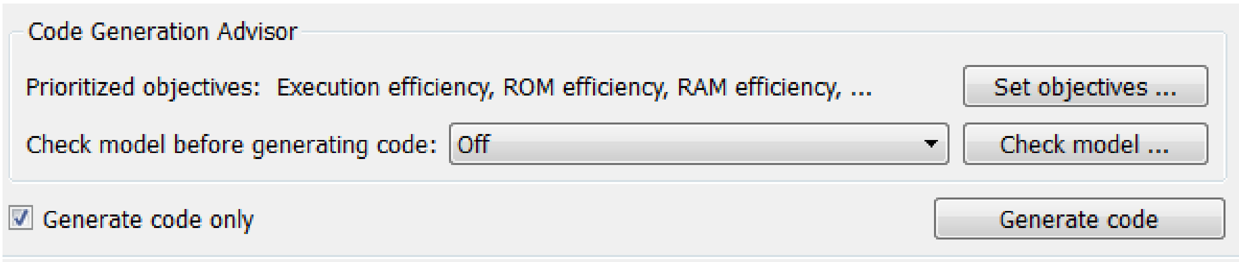


This will set the conflicting parameters to the correct settings for code generation. Once this is done click on Run This Check will in the same window:

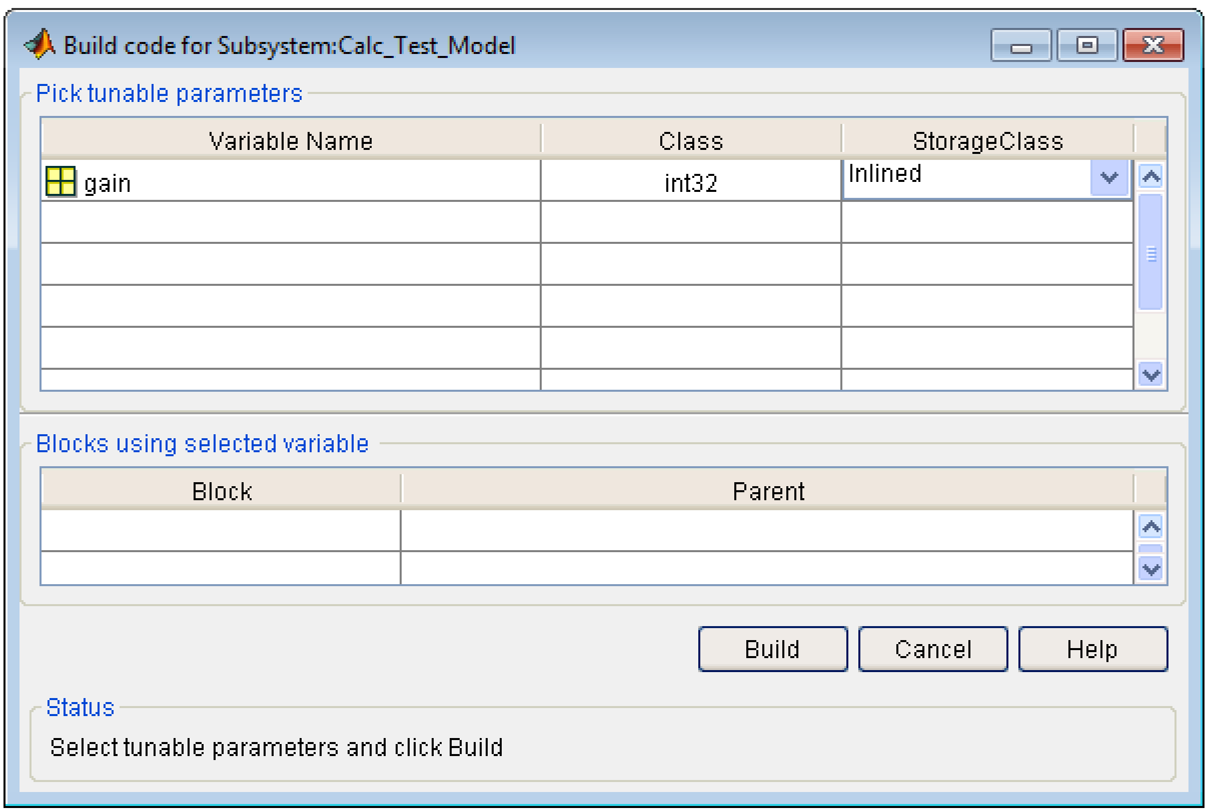


Do the same procedure for the remaining warnings if necessary based on individual model configurations. In the case of this simple example it is not necessary to fix all warnings as they are only there to prevent what could be future issues but not really simulation errors.

Once done here go back to the Configuration Parameters and change the "Check model before generating code" dropdown list to the "Off" setting in order to bypass the model advisor. Click Apply then OK to save these changes then close the Configuration Parameters:

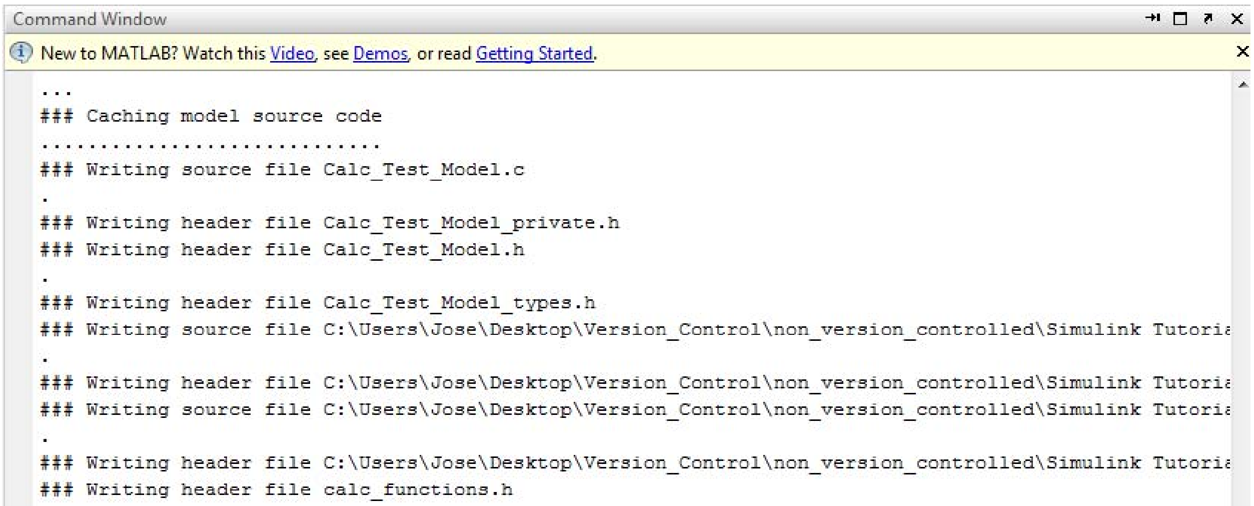


Right click on the Calc_Test_Model Subsystem and go to Code Generation > Build Subsystem. If there are not conflicts the following window should appear. (NOTE: It is possible an error message might pop-up due to an unset or incorrectly set configuration parameter option. This is normal and can be easily fixed by changing the conflicting parameter):

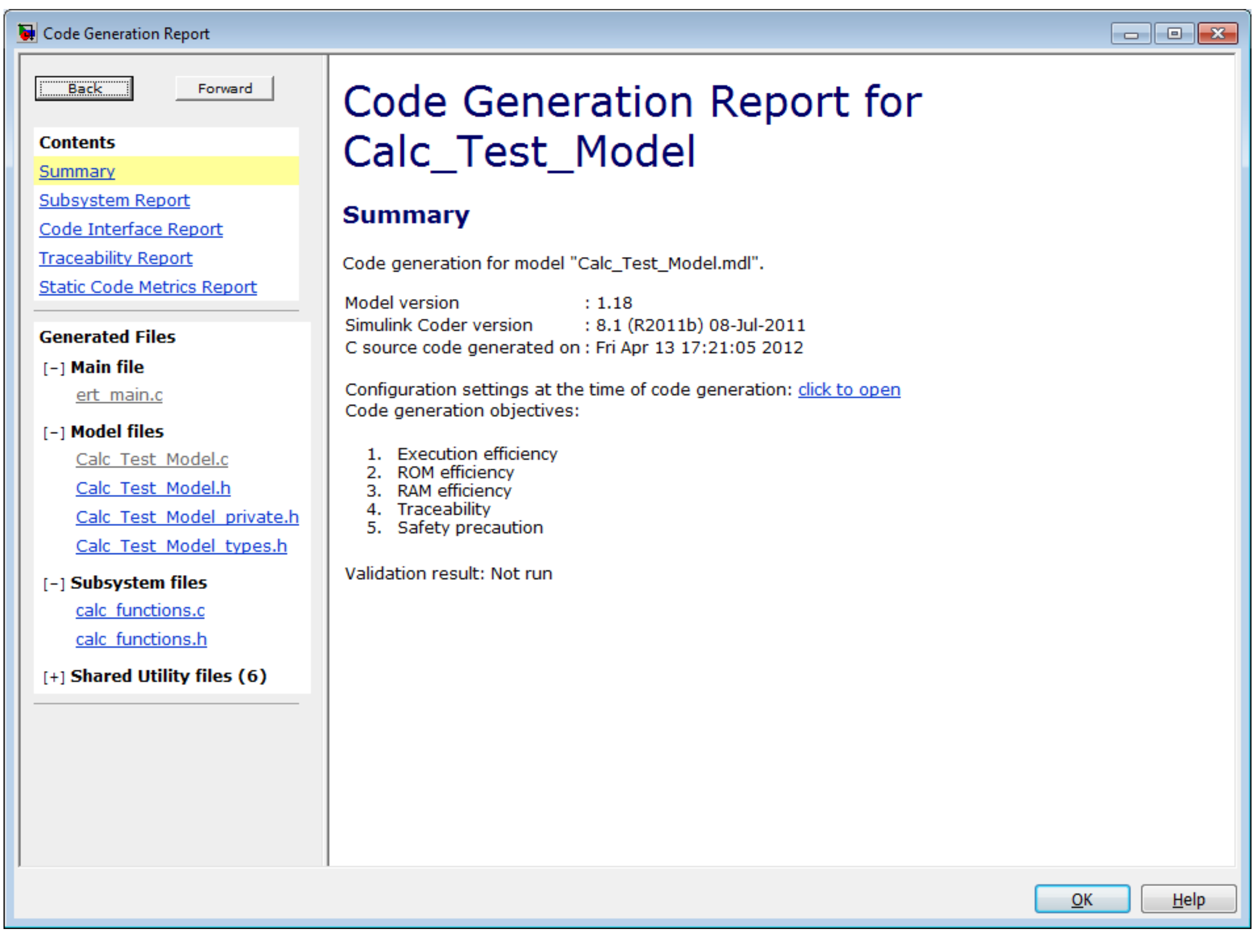


This window contains all workspace constants that are to be used in the generated code and offer storage options for the generated code. In the case of this example the variable gain can be left as "Inlined". Click on Build and the code generation process will start.

The code generation progress should be shown in the command window as seen below:

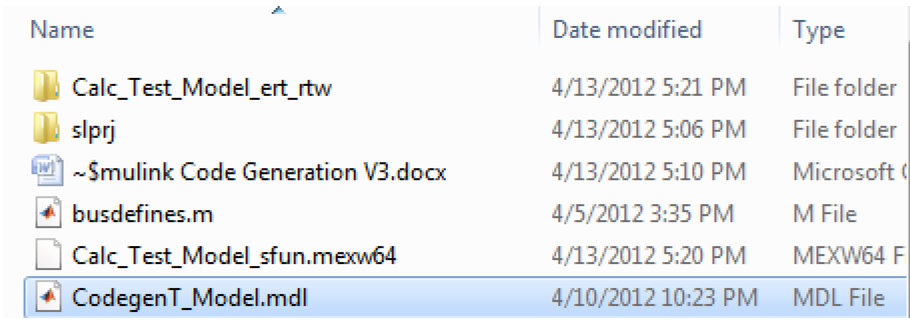


If all goes well the following Code Generation Report should show up:



From here the generated code can be evaluated and can also be traced back to the Simulink (Ref. [2]) Model. For more information on the Code Generation Report consult the MATLAB (Ref. [1]) help window.

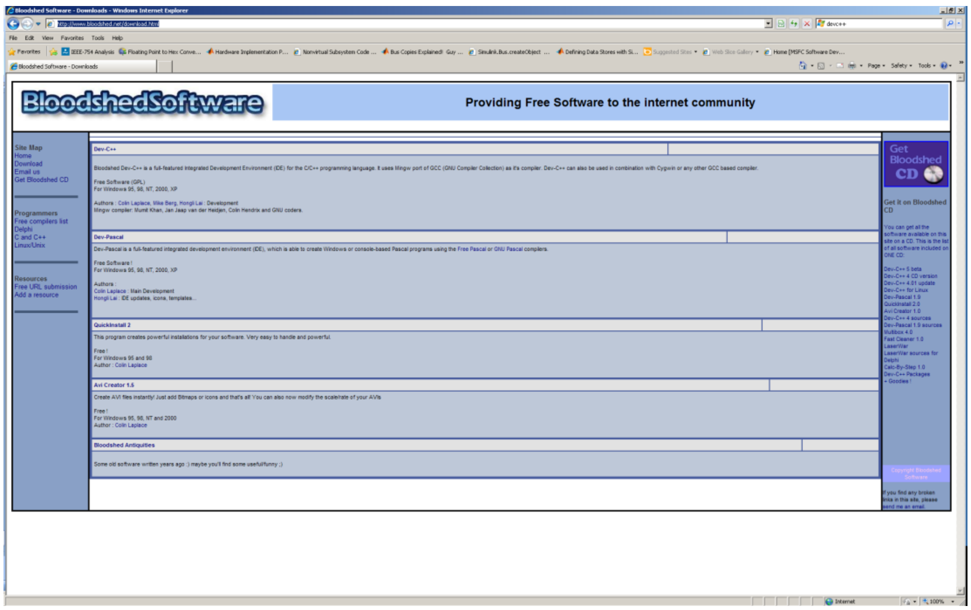
Now on the current MATLAB (Ref. [1]) path there should be two folders called Calc_Test_Model_ert_rtw and slprj respectively:



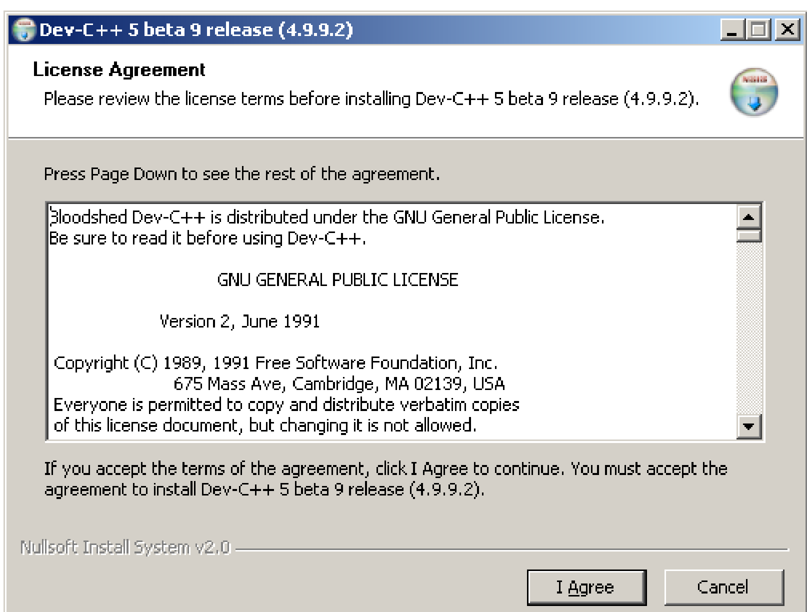
These folders contain all the generated code from the Simulink (Ref. [2]) Model; copy and paste these folders to any desired location for the next section of the tutorial where the code will be compiled and executed.

8. Compiling and Executing generated code

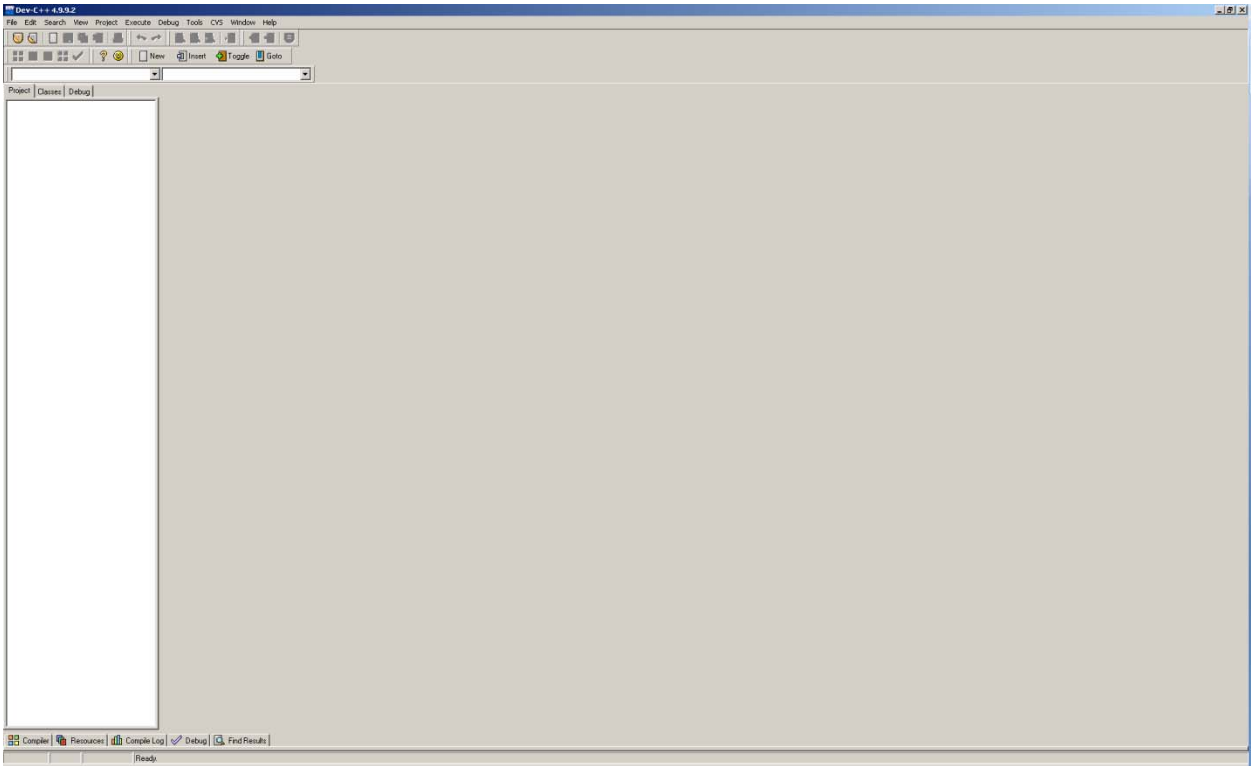
Now that the code is generated the next step is to test the code by compiling it and creating an executable to run it. For this section of the tutorial a C IDE and compiler are required. This tutorial will use Bloodshed Software’s Dev-C++ IDE [2] but these steps can be done in any IDE with any compiler. Dev-C++ (Ref. [3]) can be downloaded from Bloodshed Software’s webpage: http://www.bloodshed.net/download.html



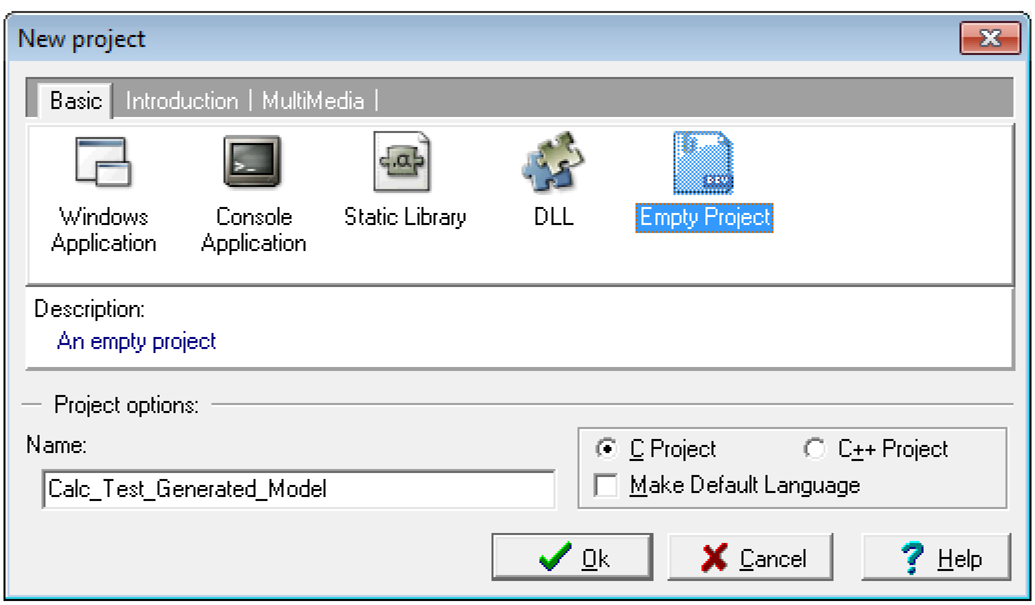
Download the Dev-C++ (Ref. [3]) setup and install it by following the on-screen prompts:



Once finished with the installation open up the Dev-C++ (Ref. [3]) IDE:

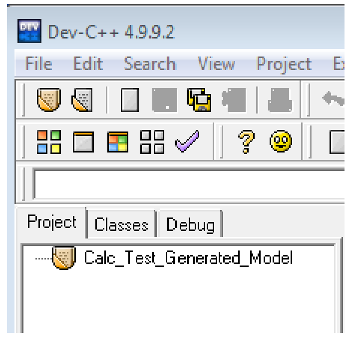


Go to the top toolbar then go to File > New > Project, afterwards select the following options: Empty Project, check C Project and Name the project Calc_Test_Generated_Model:

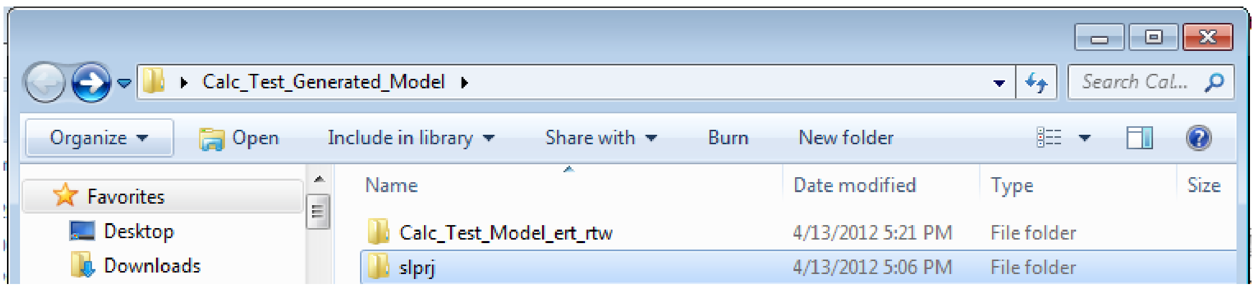


Click on Ok and a Prompt will come up asking for a location to save the project. Choose any desired location. (NOTE: It is recommended to create a new folder for this project)

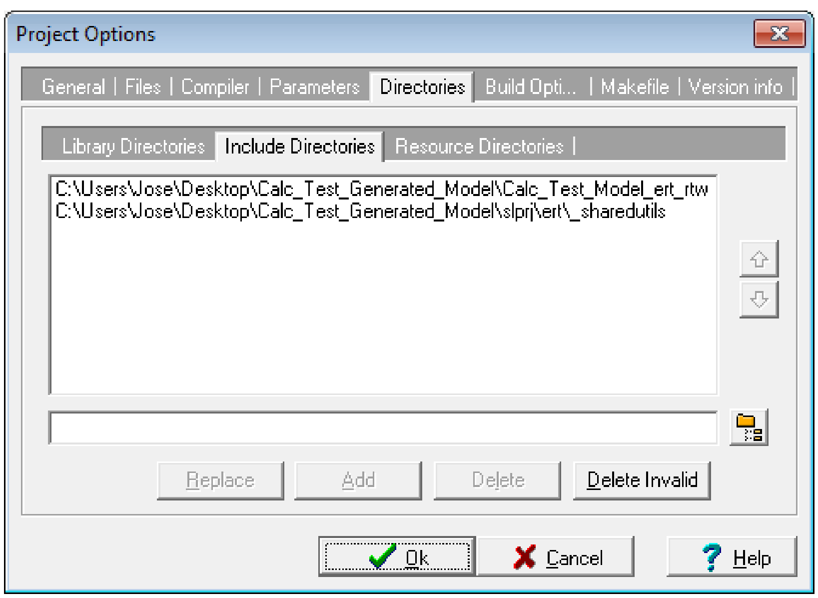
After creating the project the left hand tab will contain a root folder with the name of the project. This is where all source code will be located:



Now drag the copies of the Calc_Test_Model_ert_rtw and the slprj folders into the Calc_Test_Generated_Model project folder in the previously selected location:

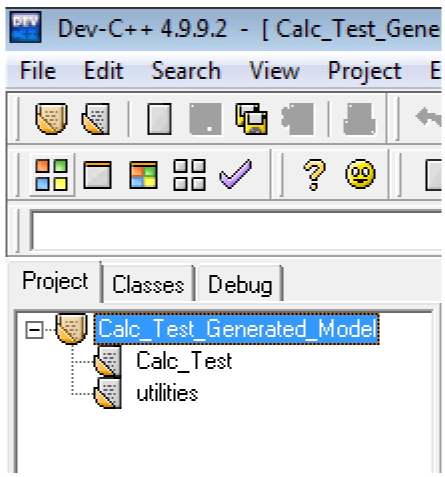


Next go to the top toolbar then go to Project > Project Options then in the Pop Up window go to the Directories Tab then inside of it go to the Include Directories Sub Tab. Add the following paths to the include directories: ".\Calc_Test_Model_ert_rtw" and ".\slprj\ert_sharedutils" (NOTE: replace the dot with the absolute path to the Calc_Test_Generated_Model folder):

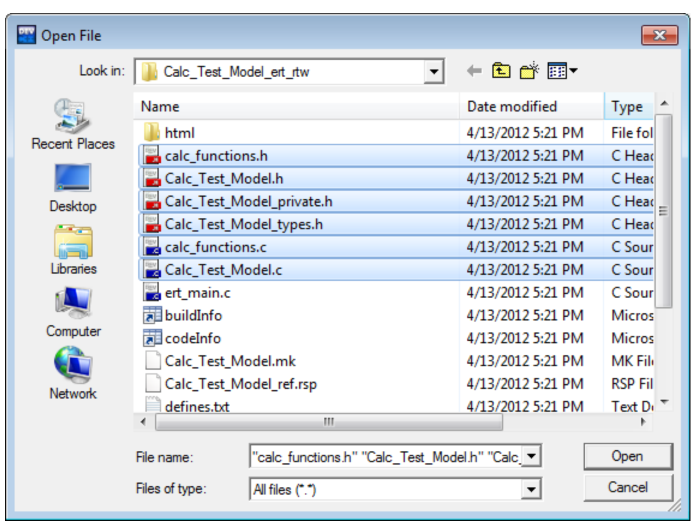


Right click on the Calc_Test_Generated_Model folder on the left hand tab and then click on "Add Folder" call the folder Calc_Test then click OK:

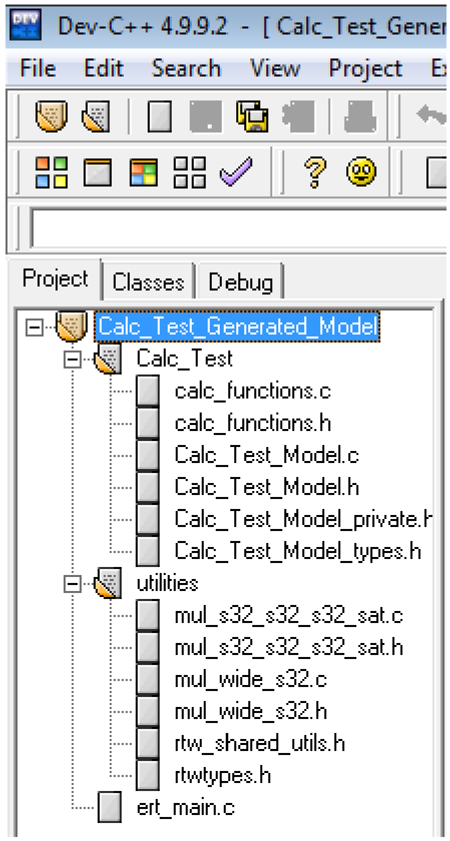
Add an additional folder to the top folder called utilities:



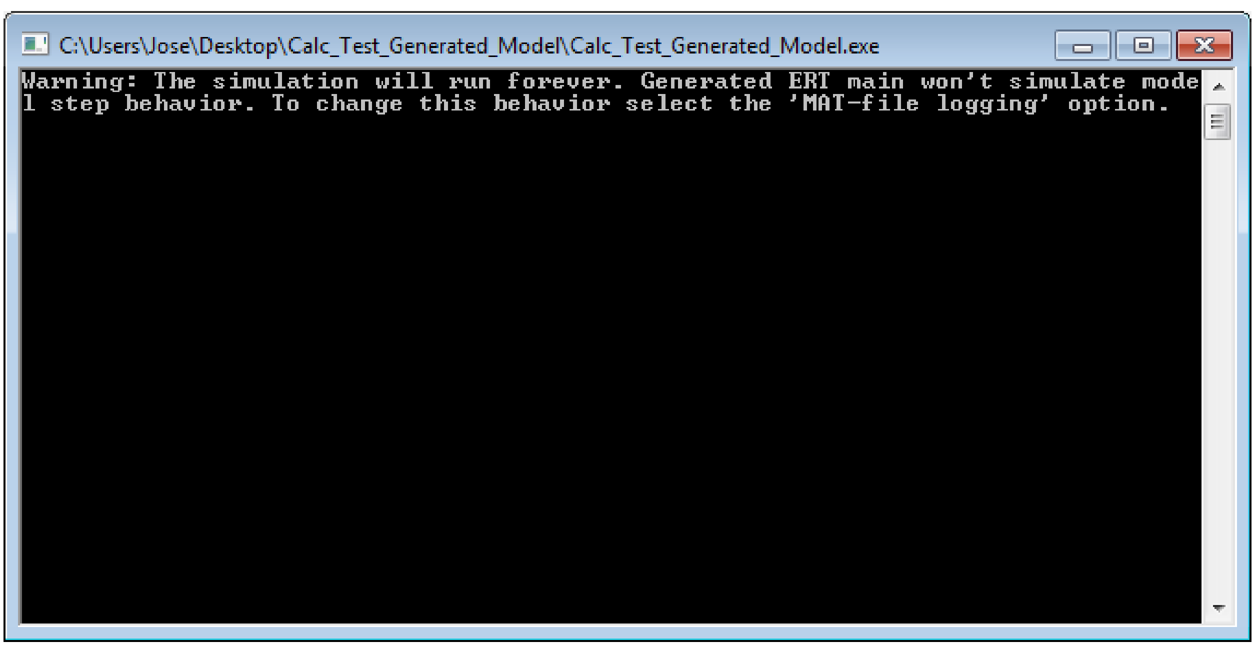
Right click on Calc_Test and then click on "Add to project". Browse to the Calc_Test_Model_ert_rtw folder then add all .c and .h files excluding the ert_main.c. Repeat this for the utilities folder by going to slprj\ert_sharedutils. Finally right click on the top Calc_Test_Generated_Model folder and add the ert_main.c from the Calc_Test_Model_ert_rtw folder:



After adding everything the folders should contain the following:



Go to the top toolbar then click on Execute > Compile & Run and if there are no problems the following output should appear:



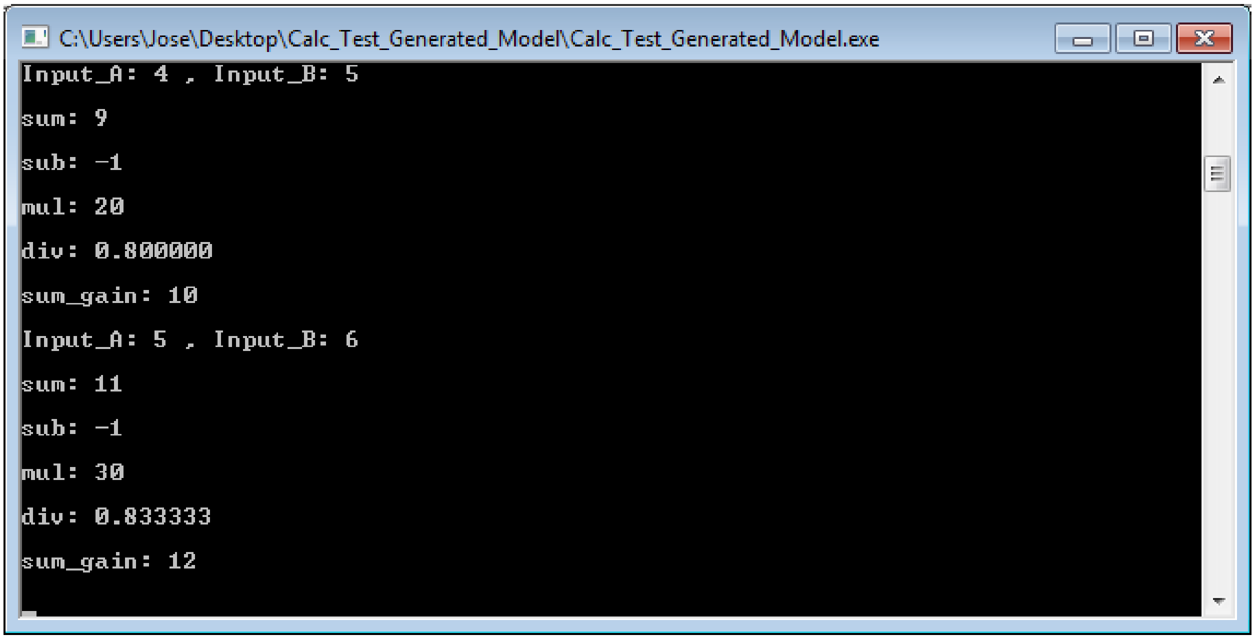
Now that the code is up and running; open the ert_main.c and make the following modifications to the main function:

```
#if 0
  printf("Warning: The simulation will run forever. "
         "Generated ERT main won't simulate model step behavior. "
         "To change this behavior select the 'MAT-file logging' option.\n");
  fflush((NULL));
  while (rtmGetErrorStatus(0) == (NULL)) {
    /*  Perform other application tasks here */
  }
  #else
  int32_T Input_A = 1;
  int32_T Input_B = 2;

  while(1)
  {
        /* Pass input data */
        Calc_Test_Model_U_Input_A = Input_A;
        Calc_Test_Model_U_Input_B = Input_B;

        /* Run Subsystem */
        Calc_Test_Model_step(&Calc_Test_Model_DWork, Calc_Test_Model_U_Input_A,
                             Calc_Test_Model_U_Input_B, &Calc_Test_Model_Y_Out1);

        /* View Output */
```

```
        printf("Input_A: %ld , Input_B: %ld\n\n"
            "sum: %ld\n\n"
            "sub: %ld\n\n"
            "mul: %ld\n\n"
            "div: %lf\n\n"
            "sum_gain: %ld\n\n"
            ,
            Calc_Test_Model_U_Input_A,
            Calc_Test_Model_U_Input_B,
            Calc_Test_Model_Y_Out1.sum,
            Calc_Test_Model_Y_Out1.sub,
            Calc_Test_Model_Y_Out1.mul,
            Calc_Test_Model_Y_Out1.div,
            Calc_Test_Model_Y_Out1.sum_gain);

        Input_A++;
        Input_B++;

        sleep(1000);

    }
    #endif
```

The modifications to the code add the Inputs A and B initialized to 1 and 2 respectively. Then an infinite while loop is started. Inside this loop the Inputs are passed to their corresponding structure elements located in the input variables for the model step function denoted by the _U (e.g. Calc_Test_Model_U_Input_A), next the step function, denoted _step (e.g. Calc_Test_Model_step) is called to run the model. After calling the step function the values are now in the output data structure denoted by the _Y (e.g.:Calc_Test_Model_Y_Out1) and then these can be accessed globally for any purpose.

The output data can then be viewed with the printf function as shown below:



9. Comparing output of MATLAB with that of the generated code

Once the code is generated and running there is a couple of ways to determine if the output is correct:

- Simple observation of the simulation output can be used in the case of simple calculations as those carried out in this example.
- In the case of more complex calculations a text file can be generated and plotted to compare to output data from the simulation model.
- Text files can be loaded into MATLAB (Ref. [1]) then the difference between these two files can be plotted showing differences in output for each sample.

References

Computer Software

[1] MATLAB, The MathWorks, Inc., Software Package, Ver. R2011b, Natick, Massachusetts, 2011.

[2] Simulink, The MathWorks, Inc., Software Package, Ver. R2011b, Natick, Massachusetts, 2011.

[3] Dev-C++, Bloodshed Software, Software Package, Ver. 5, 2005.

Appendix A Hardware Implementation Table

Key:	
	float and double (not listed) always equal 32 and 64, respectively
	pointer (not listed) matches the value of int
	Rounds to = Signed integer division rounds to
	Shift right = Shift right on a signed integer as arithmetic shift

Device vendor / Device type	Number of bits					Largest atomic size		Byte ordering	Rounds to	Shift right
	char	short	int	long	native	int	float			
Generic										
Unspecified (assume 32-bit Generic) (default)	8	16	32	32	32	x	x	Unspecified	x	Set
Custom	x	x	x	x	x	x	x	x	x	x
16-bit Embedded Processor	8	16	16	32	16	x	x	x	x	Set
32-bit Embedded Processor	8	16	32	32	32	x	x	x	x	Set
32-bit Real Time Simulator	8	16	32	32	32	x	x	x	x	Set
32-bit x86	8	16	32	32	32	x	x	Little	Zero	Set

Key:	float and double (not listed) always equal 32 and 64, respectively									
	pointer (not listed) matches the value of int									
	Rounds to = Signed integer division rounds to									
	Shift right = Shift right on a signed integer as arithmetic shift									
Device vendor / Device type	Number of bits					Largest atomic size		Byte ordering	Rounds to	Shift right
	char	short	int	long	native	int	float			
compatible								Endian		
8-bit Embedded Processor	8	16	16	32	8	x	x	x	x	Set
MATLAB Host Computer	8	16	32	Host specific value (32 or 64)	Host specific value (32 or 64)	x	x	Little Endian	x	Set
AMD										
K5/K6/Athlon	8	16	32	32	32	x	x	Little Endian	x	Set
ARM Compatible										
ARM 7/8/9/10	8	16	32	32	32	Long	Float	x	x	x
ARM 11	8	16	32	32	32	Long	Double	x	x	x

Key:	float and double (not listed) always equal 32 and 64, respectively									
	pointer (not listed) matches the value of int									
	Rounds to = Signed integer division rounds to									
	Shift right = Shift right on a signed integer as arithmetic shift									
Device vendor / Device type	Number of bits					Largest atomic size		Byte ordering	Rounds to	Shift right
	char	short	int	long	native	int	float			
ARM Cortex	8	16	32	32	32	Char	None	x	x	x
ASIC/FPGA										
ASIC/FPGA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
Analog Devices										
Blackfin	8	16	32	32	32	Long	Double	Little Endian	Zero	Set
SHARC	32	32	32	32	32	Long	Double	Big Endian	Zero	Set
TigerSHARC	32	32	32	32	32	Long	Double	Little Endian	Zero	Set
Atmel										
AVR	8	16	16	32	8	x	x	Little Endian	Zero	Set
Freescale										

Key:	float and double (not listed) always equal 32 and 64, respectively									
	pointer (not listed) matches the value of int									
	Rounds to = Signed integer division rounds to									
	Shift right = Shift right on a signed integer as arithmetic shift									
Device vendor / Device type	Number of bits					Largest atomic size		Byte ordering	Rounds to	Shift right
	char	short	int	long	native	int	float			
32-bit PowerPC	8	16	32	32	32	Long	Double	Big Endian	Zero	Set
68332	8	16	32	32	32	x	x	Big Endian	x	Set
68HC08	8	16	16	32	8	x	x	Big Endian	x	Set
68HC11	8	16	16	32	8	x	x	Big Endian	x	Set
ColdFire	8	16	32	32	32	x	x	Big Endian	Zero	Set
DSP563xx (16-bit mode)	8	16	16	32	16	x	x	x	x	Set
HC(S)12	8	16	16	32	16	x	x	Big Endian	x	Set
MPC52xx, MPC5500, MPC55xx, MPC5xx, MPC7400, MPC7xxx, MPC82xx, MPC83xx,	8	16	32	32	32	Long	Double	x	Zero	Set

<table>
<tr><td rowspan="4">Key:</td><td colspan="10">float and double (not listed) always equal 32 and 64, respectively</td></tr>
<tr><td colspan="10">pointer (not listed) matches the value of int</td></tr>
<tr><td colspan="10">Rounds to = Signed integer division rounds to</td></tr>
<tr><td colspan="10">Shift right = Shift right on a signed integer as arithmetic shift</td></tr>
<tr><td rowspan="2">Device vendor / Device type</td><td colspan="5">Number of bits</td><td colspan="2">Largest atomic size</td><td rowspan="2">Byte ordering</td><td rowspan="2">Rounds to</td><td rowspan="2">Shift right</td></tr>
<tr><td>char</td><td>short</td><td>int</td><td>long</td><td>native</td><td>int</td><td>float</td></tr>
<tr><td>MPC86xx, MPC8xx</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>MPC85xx</td><td>8</td><td>16</td><td>32</td><td>32</td><td>32</td><td>Long</td><td>Float</td><td>x</td><td>Zero</td><td>Set</td></tr>
<tr><td>S12x</td><td>8</td><td>16</td><td>16</td><td>32</td><td>16</td><td>x</td><td>x</td><td>Big Endian</td><td>x</td><td>Set</td></tr>
<tr><td colspan="11">Infineon</td></tr>
<tr><td>C16x, XC16x</td><td>8</td><td>16</td><td>16</td><td>32</td><td>16</td><td>x</td><td>x</td><td>Little Endian</td><td>Zero</td><td>Set</td></tr>
<tr><td>TriCore</td><td>8</td><td>16</td><td>32</td><td>32</td><td>32</td><td>x</td><td>x</td><td>Little Endian</td><td>x</td><td>Set</td></tr>
<tr><td colspan="11">Intel</td></tr>
<tr><td>8051 Compatible</td><td>8</td><td>16</td><td>16</td><td>32</td><td>8</td><td>x</td><td>x</td><td>x</td><td>x</td><td>Clear</td></tr>
<tr><td>x86/Pentium</td><td>8</td><td>16</td><td>32</td><td>32</td><td>32</td><td>x</td><td>x</td><td>Little Endian</td><td>x</td><td>Set</td></tr>
</table>

Key:	float and double (not listed) always equal 32 and 64, respectively									
	pointer (not listed) matches the value of int									
	Rounds to = Signed integer division rounds to									
	Shift right = Shift right on a signed integer as arithmetic shift									
Device vendor / Device type	Number of bits					Largest atomic size		Byte ordering	Rounds to	Shift right
	char	short	int	long	native	int	float			
Microchip										
PIC18	8	16	16	32	8	x	x	Little Endian	Zero	Set
dsPIC	8	16	16	32	16	x	x	Little Endian	Zero	Set
NEC										
V850	8	16	32	32	32	x	x	x	x	x
Renesas										
M16C	8	16	16	32	16	x	x	Little Endian	x	x
M32C	8	16	x	32	x	x	x	Little Endian	x	x
R8C/Tiny	8	16	16	32	16	x	x	Little Endian	x	x
SH-2/3/4	8	16	32	32	32	x	x	x	x	x

Key:	float and double (not listed) always equal 32 and 64, respectively									
	pointer (not listed) matches the value of int									
	Rounds to = Signed integer division rounds to									
	Shift right = Shift right on a signed integer as arithmetic shift									
Device vendor / Device type	Number of bits					Largest atomic size		Byte ordering	Rounds to	Shift right
	char	short	int	long	native	int	float			
SGI										
UltraSPARC IIi	8	16	32	32	32	x	x	Big Endian	x	Set
STMicroelectronics										
ST10/Super10	8	16	16	32	16	x	x	Little Endian	Zero	Set
Texas Instruments										
C2000	16	16	16	32	16	Int	None	x	Zero	Set
C5000	16	16	16	32	16	Int	None	Big Endian	Zero	Set
C6000	8	16	32	40	32	Int	None	x	Zero	Set
MSP430	8	16	16	32	16	x	x	Little Endian	Zero	Set
TMS470	8	16	32	32	x	x	x	x	x	x

Table 2: Hardware Implementation Complete